R.W. Hockney
C.R. Jesshope

CALCULATOARE PARALELE

arhitectură programare algoritmi

Editura Tehnică

Prefață la prima ediție

Anii '80 vor fi probabil decada calculatorului paralel, iar această carte are tocmai scopul de a realiza o introducere în acest domeniu. Deși, începînd cu anii '50, multe calculatoare au înglobat elemente de operare paralelă sau concurentă, abia prin 1974—5 au apărut primele calculatoare proiectate special pentru utilizarea paralelismului cu scopul prelucrării eficiente a vectorilor sau masivelor de numere. Aceste calculatoare fie executau în paralel diversele subfuncții ale unei operații aritmetice în același mod cu o linie de asamblare industrială (calculatoare pipeline ca CDC STAR și TIASC), sau multiplicînd unități aritmetice complete (masive de procesoare ca ILLIAC IV). Aceste proiecte timpurii au pus numeroase probleme, dar prin 1980 mai mulți producători importanți ofereau calculatoare paralele pe bază comercială, deosebite de proiectele de cercetare anterioare. Principalele exemple sînt CRAY-1 (prima instalare în 1976) și CDC CYBER 205, calculatoare pipeline, ICL DAP și Burroughs BSP, masive de procesoare. Din păcate, de cînd am elaborat acest material, Burroughs a avut probleme la producerea calculatorului BSP și, deși s-a construit o mașină prototip, Burroughs a abandonat acest proiect. Totuși, acesta rămîne un model foarte interesant iar abandonarea sa furnizează mai multe perspective în interiorul domeniului. Calculatoarele pipeline devin populare, de asemenea, ca procesoare ce se cuplează la minicalculatoare (pentru prelucrarea semnalelor sau analiza datelor seismice) sau la microcalculatoare. Ca exemple dăm FPS AP-120B, FPS-164, Data General AP/130 și IBM 3838.

Paralelismul a fost introdus deoarece numai creșterea vitezei circuitelor nu poate produce performanța cerută. Aceasta apare evident în studiile de proiect elaborate pentru propusa Facilitate de Simulare Aeronautică Națională (NASF) a NASA de la Ames. Se dorește un calculator capabil de a executa 10^9 operații în virgulă mobilă pe secundă. Propunerea firmei CDC pentru această mașină se referă la patru unități pipeline de mare performanță, în timp ce proiectul Burroughs este un masiv de 512 unități aritmetice identice. Progresele tehnologiei de integrare pe scară foarte largă (VLSI) asigură posibilitatea realizării unor masive foarte mari de elemente procesoare (PE). ICL DAP (4096 PE) și Goodyear Aerospace MPP sînt exemple timpurii de masive de procesoare ce vor fi, probabil, dezvoltate în cursul acestor ani pentru a exploata avantajele tehnologiei VLSI. Se poate pretinde că proiectele de mai sus interesează numai marile laboratoare de cercetare științifică și că nu vor avea un impact asupra masei de utilizatori de calculatoare. Totuși, experiența arată că progresele înregistrate de arhitectura calculatoarelor produse la început pentru piața științifică au devenit mai tîrziu parte a calculatoarelor de uz general.

De aceea, se pare că un grad mai mare de paralelism (datorită cererii unei performanțe mai mari decît pot asigura circuitele electronice), împreună cu tehnologia necesară implementării acestuia (VLSI) vor face probabil din paralelismul arhitecturii calculatoarelor un domeniu de mare interes în cursul anilor '80. Scopul acestei cărți este de a explica principiile și clasificarea atît a calculatoarelor pipeline cît și a masivelor, să prezinte cum au fost implementate aceste principii în cadrul unor calculatoare de succes (CRAY-1, CYBER 205, FPS AP-120B, ICL DAP și Burroughs BSP) și să compare performanțele diverselor sisteme pentru un număr de aplicații importante (operații cu matrici, FFT, rezolvarea ecuației Poisson). Apariția arhitecturilor înalt paralele pune, de asemenea, problema proiectării unor algoritmi numerici eficienți, ca și a unor limbaje în care să se exprime acești algoritmi. Considerăm că algoritmii și limbajele sînt aspecte ale paralelismului la fel de importante cu arhitectura, de aceea le dedicăm capitole separate.

O caracteristică a tratării noastre care poate fi considerată originală este conceptul unei descrieri cu doi parametri a performanței calculatoarelor : în plus față de obișnuita performanță maximă exprimată în operații în virgulă mobilă pe secundă r_∞*, propunem lungimea performanței jumătate* $n_{1/2}$*, lungimea vectorului necesar pentru a atinge jumătate din performanța maximă. Această variabilă poate fi de asemenea interpretată ca măsurînd cantitatea de paralelism a unui calculator și variază de la* $n_{1/2} = 0$ *pentru calculatoarele seriale la* $n_{1/2} = \infty$ *pentru un masiv infinit de procesoare. În acest mod, al doilea parametru poate fi bine folosit pentru clasificarea arhitecturilor paralele, conform acestei măsuri cantitative. Deci, răspunde întrebării : cît de paralel este calculatorul meu? De egală importanță este faptul că* $n_{1/2}$ *determină cantitativ și alegerea algoritmului cel mai eficient. Deci, răspunde și la întrebarea : care este cel mai bun algoritm pentru calculatorul meu paralel? De asemenea, propunem o notație în stil algebric, care permite definirea arhitecturii unui calculator pe o linie, eliminînd astfel necesitatea unor descrieri prolixe și ajutînd la clasificarea sistemelor prin posibilitatea descrierii cu o singură formulă.*

Cea mai mare parte a acestei cărți a fost predată ca un curs la Facultatea de știința calculatoarelor a Universității Reading, denumit „Arhitectura calculatoarelor avansate". Acest curs a ajuns la 40 de lecții în ultimii 5 ani, iar lipsa unui text adecvat a motivat scrierea volumului de față. Cursul este opțional în anul III, dar poate fi folosit și ca specializare pentru anii finali (nivel MSc), sau pentru pregătirea unei teze de doctorat în domeniul calculatoarelor paralele.

Multe persoane au ajutat, prin discuții și observații, la pregătirea acestui manuscris. Printre ei dorim să-i menționăm pe colegii de la Universitatea Reading : în special Jim Craigie, John Graham, Roger Loader, John Ogden, John Roberts și Shirley Williams ; și Henry Kemhadjian de la Universitatea Southampton. Dr. Ewan Page, vice-cancelar al Universității Reading și editorul seriei, Prof. Mike Rogers de la Universitatea Bristol, au propus îmbunătățiri ale manuscrisului. De asemenea, am primit informații și fotografii de la reprezentanții unor firme, printre care dorim să le mulțumim lui Pete Flanders, David Hunt și Stewart Reddaway de la ICL Research and Advanced Development Centre, Stevenage, și lui John Smallbone de la ICL, Euston ; Prof. Dennis Parkinson de la ICL și Queen Mary College, Universitatea din Londra ; Stuart Drayton, Mick Dungworth și Jeff Taylor

de la CRAY Research, Bracknell; David Barkai şi Nigel Payne de la Control Data Corporation (UK), şi Patricia Conway, Neil Lincoln şi Chuck Purcell de la CDC Minneapolis; J.H. Austin de la Burroughs, Paoli şi G. Tillot de la Burroughs, Londra; C.T. Mickelson de la Goodyear Aerospace, Akron; şi John Harte, David Head şi Steve Markham de la Floating Point Systems, Bracknell. Domnişoara Jill Dickinson (acum doamna Contla), care a tipărit textul, a corectat multe erori.

Am dedicat această carte proiectanţilor de calculatoare, deoarece fără inspiraţia şi devotamentul lor nu am fi avut posibilitatea studiului, clasificării şi utilizării unei varietăţi atît de interesante de calculatoare. Inevitabil, proiectarea unui calculator este o muncă de echipă, dar dorim să exprimăm în mod special consideraţia noastră pentru Seymour Cray, Neil Lincoln, George O'Leary şi Stewart Reddaway, principalii proiectanţi ai calculatoarelor selectate pentru a fi analizate în prezentul volum.

R. W. HOCKNEY, C. R. JESSHOPE

1981

Prefață la ediția a II-a

În cei șapte ani care au trecut de la publicarea Calculatoarelor paralele, au intervenit suficiente schimbări care să justifice o a doua ediție. În afara evoluției arhitecturilor descrise în prima ediție (de exemplu, CYBER 205 la ETA[10], și CRAY-1 la CRAY X-MP și CRAY-2), au apărut multe calculatoare noi cu fluxuri multiple de instrucțiuni (MIMD) experimentale, iar unele chiar comercializate (de exemplu, Intel iPSC, Sequent Balance, Alliant FX/8). În plus, cipurile microprocesor (transputerul INMOS), disponibile acum, sînt proiectate pentru a fi conectate în rețele mari, iar multe sisteme se proiectează pe baza lor.

Deși am păstrat structura primei ediții, am inclus aceste dezvoltări prin extinderea Introducerii (prin includerea extensiilor necesare la notația arhitecturală algebrică și clasificarea proiectelor), și prin descrierea mai detailată a unor noi arhitecturi în capitolele următoare. În capitolul dedicat calculatoarelor vectoriale pipeline am inclus descrierea calculatoarelor vectoriale japoneze de succes (Fujitsu VP, Hitachi S810 și NEC SX2), ca și noua generație de calculatoare vectoriale multiple din Statele Unite (CRAY-2 și ETA[10]). În capitolul privind multiprocesoarele și masivele de procesoare am inclus Denelcor HEP, primul calculator MIMD comercializat, și Connection Machine.

Deși HEP nu mai este produs, considerăm că arhitectura sa, ce folosește partajarea unui singur pipeline pentru instrucțiuni de către multiple fluxuri de instrucțiuni, este suficient de originală și interesantă pentru a-i justifica includerea; Connection Machine reprezintă soluția opusă, a conectării unui număr foarte mare (aproximativ 65000) de procesoare simple într-o rețea.

Transputerul INMOS este primul cip comercializat care a fost conceput ca bloc de construcție pentru calculatoarele paralele; arhitectura sa și modalitățile de utilizare sînt tratate în capitolul 3. Aproximativ o duzină de calculatoare vîndute în momentul de față folosesc mai multe transputere și probabil există mai multe mii de companii din întreaga lume care studiază posibilitățile acestui cip, fie ca produs potențial, fie ca o componentă a unui sistem paralel. Interesul nu este surprinzător, dacă amintim că cîteva zeci de transputere T800 sînt capabile să atingă performanțele calculatorului CRAY-1, prezentat în prima ediție a acestei cărți. Producerea acestui cip este simptomatică pentru dezvoltările ce au intervenit în ultimii 5—10 ani în tehnologia semiconductorilor; densitatea porților s-a dublat la fiecare doi ani, iar frecvența ceasului a crescut cu 50% în aceeași perioadă. Nu este nici un semn că acest progres se va reduce semnificativ. Aceste evoluții, împreună cu implicațiile lor, sînt tratate în capitolul 6.

De la prima ediție, progresele în domeniul limbajelor paralele sînt reprezentate de includerea standardului FORTRAN 8X pentru exprimarea calculelor vectoriale, ce va fi disponibil pe calculatoarele vectoriale, CMLISP, un limbaj de programare pentru Connection Machine, și OCCAM, limbajul de programare pentru transputerul INMOS.

Apariția calculatoarelor MIMD a impus definirea parametrilor de performanță care iau în considerare costul sincronizării multiplelor fluxuri de instrucțiuni, ca și costul comunicațiilor între procesoare. La definirea parametrilor ($s_{1/2}$ pentru sincronizare, $f_{1/2}$ pentru comunicație) am urmat filosofia parametrilor (r_∞, $n_{1/2}$) propuși în prima ediție pentru caracterizarea comportării unui pipeline vectorial, și care sînt acum cunoscuți și adoptați pe scară largă. Cu alte cuvinte, am ales parametrii care pot fi legați de anumite proprietăți ale programului utilizatorului (de exemplu, cantitatea de operații aritmetice într-un bloc de cod paralelizat). Aceasta conduce la dezvoltarea metodelor de analiză ai algoritmilor ce se bazeazăpe acești parametri, prezentate în capitolul 5.

Ar lua prea mult spațiu să mulțumim tuturor celor care ne-au ajutat cu informații și fotografii pentru noua ediție, dar în mod special am dori să le mulțumim lui : David Dent (ECMWF); Paul Elstone, John Larson și William White (Cray Reserarch); Shaun Powell, P.J. Elms, Alain Hochedez și Jean-Claude Lignac (CDC); Meg Saline și Cliff Arnold (ETA Systems); David Snelling (ex Denelcor) și Ian Curington (ex FPS); David May (INMOS); Geoff Manning (AMT Ltd); Leon Bentley, Andrew Rushton Jimmy Stewart, Adriano Cruz, Russel O'Gorman, Gadge Panesar, Ernest Ng și Charles Askew (Universitatea Southampton).

O carte de calculatoare poate să devină cu ușurință o descriere plictisitoare a unui șir fără sfîrșit de arhitecturi diferite, și s-ar putea ca unii cititori să considere că am căzut în această capcană. De aceea, ca și la prima ediție, am încercat să facem prezentarea într-o perspectivă istorică, într-un cadru teoretic al unei notații, clasificări și teorii a performanței care, sperăm noi, se vor aplica chiar cînd calculatoarele descrise vor deveni istorie trecută. Sperăm că în acest mod noua ediție va rămîne la fel de populară ca un text de curs, cum a fost prima ediție, ce prezenta principii și metodologii dar și fapte depășite între timp. Cu toate acestea, am prezentat multe din calculatoarele, limbajele și algoritmii care vor fi folosite în următorii 5 ani de către utilizatorii de calculatoare, specialiști în domeniul calculatoarelor, sau în diverse discipline (de exemplu, fizica, chimia, biologia și ingineria). De aceea, ne așteptăm ca această nouă ediție să fie la fel de folositoare unor grupe diferite de oameni care doresc să atingă performanțele cele mai bune cu numeroasele și variatele arhitecturi de calculatoare paralele disponibile.

R. W HOCKNEY
C. R. JESSHOPE

(Universitățile Reading și Southampton, august 1987)

Cuprins

1.

INTRODUCERE

Principala inovație în proiectarea calculatoarelor comerciale a anilor 80 este introducerea paralelismului pe scară largă. În acest capitol introductiv vom prezenta istoria acestui concept (§ 1.1), vom expune principiile utilizate pentru clasificarea calculatoarelor paralele (§ 1.2) și vom caracteriza performanțele lor relative (§ 1.3). Pentru utilizarea eficientă a calculatoarelor paralele, programatorul trebuie să cunoască organizarea sistemului definită, astfel, ca arhitectura sa : ne referim la numărul și tipul procesoarelor, modulele de memorie și canalele de I/E și cum sînt acestea interconectate și comandate. Capitolele 2 și 3 descriu arhitectura a două clase principale de calculatoare paralele, *calculatoarele pipeline* și, respectiv, *masivele de procesoare* (array). Ar fi imposibil să descriem toate sistemele din această categorie ; în schimb, am selectat principalele calculatoare comerciale, cu un număr semnificativ de vînzări, și care diferă suficient între ele pentru a ilustra soluții diferite la aceeași problemă. În capitolul 2 sînt descrise calculatoarele pipeline CRAY X-MP, CRAY-2, CYBER 205, ETA[10] și, FPS 164/MAX. În capitolul 3 sînt descrise masivele de procesoare și sistemele multiprocesor ICL DAP, Burroughs BSP, Denelcor HEP și Connection Machine. În capitolul 6, unde discutăm dezvoltările viitoare, comparăm caracteristicile tehnologiilor bipolară și MOS, examinăm aspectele tehnologice ridicate de proiectarea calculatoarelor paralel și analizăm potențialul oferit de integrarea wafer-scale.

Apariția tehnologiilor de integrare pe scară foarte largă (VLSI), ce a culminat cu apariția microprocesorului pe un singur cip, a condus la propunerea unui număr mare de proiecte de calculatoare unde, la rezolvarea unei probleme, cooperează mai multe fluxuri de instrucțiuni. Astfel de calculatoare, cu mai multe fluxuri de instrucțiuni sau MIMD, sînt trecute în revistă în cap. 1 în § 1.1.8., clasificate în § 1.2.5, iar unul dintre ele, Denelcor HEP, este descris în detaliu în cap. 3 § 3.4.4.).

Noile caracteristici arhitecturale solicită limbaje și algoritmi numerice noi pentru a exploata avantajele oferite. Deoarece calculatoarele paralele sînt proiectate să lucreze cel mai eficient cu liste mono-sau-bi-dimensionale de informații, se introduce în limbajele calculatoarelor conceptul matematic de vector sau matrice, iar algoritmii sînt analizați în funcție de capacitatea lor de a prelucra vectori. Cum un limbaj bun influențează mult ușurința programării, capitolul 4 este dedicat *limbajelor paralele* analizate din punctul de vedere al paralelismului implicit (§4.2), paralelismu-

lui structurii (§4.3) şi paralelismului procesului (§4.4). Utilitatea calculatoarelor paralele depinde de inventarea sau selectarea unor *algoritmi paraleli* adecvaţi şi de aceea în capitolul 5 stabilim principiile pentru măsurarea performanţelor algoritmilor pentru un calculator paralel. Apoi, acestea sînt aplicate pentru selectarea algoritmilor pentru recurenţe, înmulţirea matricilor, transformate, rezolvarea ecuaţiilor liniare tridiagonale şi a unele ecuaţii diferenţiale parţiale.

Deşi prezentarea noastră se bazează pe analiza situaţiei la nivelul *aproximativ* al anului 1987, încercăm să stabilim principii pe care utilizatorul le va întîlni şi folosi şi în viitor. Obiectivul nostru este de a permite cititorului să evalueze noile proiecte prin clasificarea arhitecturii lor şi caracterizarea performanţelor, iar de aici să poată selecta algoritmi şi limbaje adecvate pentru rezolvarea unor probleme specifice.

1.1. Istoria paralelismului şi a supercalculatoarelor

Prezentarea noastră istorică va fi în primul rînd un studiu al influenţei paralelismului asupra arhitecturii calculatoarelor de mare performanţă, situate la extremitatea superioară din punct de vedere al performanţelor, proiectate pentru rezolvarea unor probleme ştiinţifice sau inginereşti dificile (de exemplu, rezolvarea ecuaţiilor cu derivate parţiale în timp tridimensionale). În aceste cazuri se cere o performanţă maximă la execuţia operaţiilor aritmetice în virgulă mobilă, atinsă printr-o combinaţie între noile tehnologii şi introducerea paralelismului în arhitectura calculatoarelor. Calculatoarele ce satisfac condiţia de mai sus sînt cunoscute ca *supercalculatoare*, de aceea acest subcapitol poate fi considerat la fel de bine ca un istoric al supercalculatoarelor.

Vom prezenta istoria introducerii paralelismului în arhitectura calculatoarelor adoptînd criteriul timpului cerut pentru execuţia unei operaţii aritmetice simple, de exemplu o înmulţire în virgulă mobilă, începînd cu primul calculator comercializat, UNIVAC 1, apărut în anul 1951. Rezultatul este ilustrat în figura 1.1 şi demonstrează o creştere de 10 ori a vitezei de calcul la fiecare 5 ani. Creşterea senzaţională a vitezei de calcul a fost posibilă prin combinarea progreselor tehnologice cu introducerea progresivă a paralelismului la toate nivelurile arhitecturii calculatoarelor.

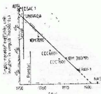


Fig. 1.1. Istoria modului de creştere a vitezei de calcul începînd cu anul 1950, ce ilustrează o creştere de 10 ori la fiecare 5 ani.

Prima generaţie de calculatoare, a anilor '50, folosea tuburi electronice caracterizate de un timp de întîrziere /poartă* de aproximativ 1 microsec. Acestea au fost înlocuite prin anii '60 cu tranzistoare cu germaniu (timp de întîrziere/poartă de aproximativ 0,3 microsec.) care definesc calculatoarele din *generaţia a doua*, ca IBM 7090. În jurul anului

* Timpul de întîrziere /poartă reprezintă intervalul de timp necesar propagării semnalului de la intrarea unei porţi logice la intrarea următoarei porţi .(Vezi, de ex., Turn 1974, p. 147) Valorile indică numai ordinul de mărime al timpului de întîrziere.

1965 a început introducerea circuitelor integrate pe scară redusă (SSI), cu cîteva porţi pe un circuit şi un timp de întîrziere/poartă de 10 ns. Acest proces a continuat pînă prin 1975, cînd s-au obţinut timpi de propagare cu ceva mai mici ca 1 ns. Tehnologia MOS a permis o densitate de porţi pe circuit mult mai mare, deşi cu o scădere a vitezei semnalului de 5 pînă la 10 ori. Pe la începutul anilor '80 au invadat piaţa microprocesoarele, cu o viteză şi capacitate aproximativ egale cu a primei generaţii de calculatoare, dar realizate pe un singur cip de siliciu de cîţiva mm pătraţi. Acest progres ingineresc a făcut, în mod evident, posibilă implementarea unor arhitecturi cu un grad mare de paralelism, care anterior erau studiate numai teoretic.

Analizînd perioada 1950—1975 se observă că viteza de bază a componentelor, măsurată ca inversul timpului de întîrziere/poartă, a crescut de aproximativ 1000 ori, în timp ce performanţa calculatoarelor, măsurată ca inversul timpului de execuţie a unei înmulţiri, au crescut de aproximativ 100000 ori. Creşterea suplimentară a fost posibilă prin perfecţionarea arhitecturii, în principal prin introducerea paralelismului, care este subiectul principal al acestei cărţi. Din motive economice, diferitele tehnologii favorizează introducerea paralelismului în moduri diferite. De exemplu, în capitolul 6, discutăm influenţa tehnologiilor de integrare VLSI şi WSI asupra modului cum se va introduce paralelismul în arhitectura viitoarelor supercalculatoare ştiinţifice. Turn (1974) discută şi el în cartea sa „*Computers in the 1980's*" dezvoltarea tehnologică, inclusiv unele previziuni interesante pentru anii '80. Acelaşi subiect este tratat şi de Sumner (1982) în „*State of the Art Report : Supercomputer Systems Technology*". În continuare ne vom referi la aspectele organizatorice ale introducerii paralelismului, mai mult decît la detaliile implementării în hardware.

Analiza de mai sus nu a luat în considerare posibilele suprapuneri ale operaţiilor logice şi aritmetice diferite, dar putem compara în schimb perioada ceasului (clock period), ca o măsură a vitezei dată de tehnologie, cu numărul operaţiilor aritmetice utile executate pe secundă pentru a rezolva o problemă, ca o măsură a performanţelor. Wilkes et al. (1951) menţionează un timp de execuţie de 18n ms pentru evaluarea unei serii de n termeni (2n operaţii aritmetice) pentru EDSAC 1 (*Wilkes şi Renwick 1949*), care avea o perioadă a ceasului de 2 microsec. Aceasta înseamnă o viteză medie de 100 operaţii aritmetice/s pe care o putem compara cu cea de 130 mil. op. aritmetice/s a calculatorului CRAY-1 (*Russell 1978*) realizată la înmulţirea matricelor. Deoarece CRAY-1 are o perioadă de tact de 12,5 ns şi creşterea performanţelor în ultimele trei decade este de 10^6, din aceasta numai un factor de aproximativ 160 poate fi atribuit progreselor tehnologice. De asemenea, trebuie remarcată creşterea calităţii operaţiilor aritmetice, de la operanzi în virgulă fixă pe 36 de biţi, în cazul lui EDSAC 1, la operanzi în virgulă mobilă de 64 de biţi la CRAY-1.

Arhitectura generală a primei generaţii de calculatoare este descrisă ca serială şi respectă ideile fundamentale ale calculatoarelor cu program memorat prezentate de Burks et al. (1946) şi cunoscute ca organizare von Neumann. Un astfel de calculator este format dintr-un dispozitiv de I/E , o memorie unică pentru stocarea instrucţiunilor şi operanzilor, o singură unitate de control pentru interpretarea instrucţiunilor şi o unitate aritmetico-logică pentru prelucrarea datelor. Ultimele două unităţi for-

mează unitatea centrală sau CPU. Caracteristica importantă a acestei organizări este execuția secvențială a tuturor operațiilor, (memory fetch, store, operații aritmetice și logice, operații de I/E). Paralelismul se referă la capacitatea de a suprapune sau executa simultan mai multe operații.

În § 1.2 sînt descrise pe larg căile de introducere a paralelismului în arhitectura calculatoarelor. Pe scurt acestea sînt:

a) *Pipelining* — creșterea performanțelor unităților aritmetice sau de comandă prin utilizarea unor tehnici similare liniei de asamblare;

b) *Funcțională* — mai multe unități independente realizează funcții diferite (logice aritmetice) simultan cu date diferite;

c) *Masiv de procesoare* — un masiv de elemente procesoare (PE) identice, care realizează simultan aceeași operație sub aceeași comandă asupra unor date diferite stocate în memorii locale;

d) *Multiprocesare* — mai multe procesoare, fiecare executînd instrucțiunile proprii și comunicînd, în general, printr-o memorie comună, sau conectate într-o rețea.

Desigur, proiectele individuale pot îmbina unele sau toate tipurile de organizare enunțate. De exemplu, un masiv de procesoare poate avea unitățile aritmetice pipeline sau o unitate funcțională a unui calculator cu mai multe unități poate fi un masiv de procesoare.

Spre sfîrșitul anilor '70 proiectele multiprocesor au implementat metode de conectare a mai multor calculatoare independente pentru a maximiza performanțele ansamblului. Această soluție este desigur importantă, dar nu reprezintă scopul acestei cărți. În consecință, vom prezenta exemple de sisteme multiprocesor numai cînd apar ca exemple de paralelism într-un calculator. Referitor la aceste sisteme în contextul mai larg al performanțelor se pot consulta *Enslow* (*1974, 1977*), *Infotech* (*1976*) *și Jones și Schwarz* (*1980*). Apariția în jurul anului 1975 a microprocesorului ieftin face posibilă implementarea sistemului cu mai multe microprocesoare ce cooperează la rezolvarea unei probleme, sisteme ce vor deveni, probabil importante în anii '80. S-au propus și realizat deja sisteme experimentale (*Satyanarayanan* (*1980*), *Hockney* (*1985 b, d*) și §1.1.8). O tratare excelentă a paralelismului, cu toate aspectele sale (unități aritmetice, organizarea memoriei, programare, I/E, multiprocesare și multiprogramare) este realizată de *Lorin* (*1972*) în cartea sa „*Parallelism in Hardware and Software : Real and Apparent Concurrency*". Hwang și Briggs (1984) au realizat în lucrarea lor „*Computer Architecture and Parallel Processing*" un studiu comprehensiv al tuturor tipurilor de arhitecturi paralele. Supercalculatoarele sînt considerate în mod particular de către Lazou (1986) în volumul „*Supercomputers and their Use*".

Noi nu încercăm o prezentare completă a istoriei paralelismului, ci numai a progreselor care au marcat pași importanți în introducerea lui. Nu putem menționa toate mașinile sau lucrările publicate. În schimb, am ales acele mașini care fie că au avut succes comercial, fie au avut o influență importantă asupra domeniului. Similar sînt menționate numai unele lucrări științifice cu o mare influență. În figura 1.2 sînt ilustrate principalele direcții și conexiuni privind introducerea paralelismului în arhitectura calculatoarelor. Vom trata separat în continuare principalele direcții de dezvoltare.

Cititorii interesați de primele începuturi ale calculatoarelor, înainte de 1950, pot consulta colecția de articole editată de Randell (1975) și intitulată „*The Origins of Digital Computers : Selected papers*" și cartea lui Hartree (1950) „*Calculating Instruments and Machines*". *Metropolis, Howlett și Rota* (*1980*) au reunit sub titlul „*A History of Computing in the Twentieth Century*" o serie de eseuri scrise de pionieri în domeniul calculatoarelor. Se tratează atît activitatea din timpul războiului în S.U.A. și Marea Britanie, cît și cercetări efectuate în U.R.S.S., Japonia, Germania și Cehoslovacia. Kuck (1978) realizează în cartea „*The Structure of Computers and Computations*" o descriere foarte plăcută a istoriei proiectării calculatoarelor de la Babbage pînă la începutul anilor '50. Este inclusă și o descriere foarte interesantă a influențelor reciproce între personalitățile implicate în etapa inițială de dezvoltare a calculatoarelor numerice, ca și a problemelor inginerești cu care au fost confruntate. O analiză istorică valoroasă este cea a lui Rosen (1969) care descrie evoluția calculatoarelor de la ENIAC în 1946 la CDC 7600 în 1968. Informații detaliate despre arhitectura multor calculatoare pe care le vom menționa se găsesc în Bell și Newell (1971) „*Computer Structures*; *Readings and Examples*". Această carte acoperă perioada 1950—1970 și cuprinde detalii ce nu pot fi găsite altundeva. Această carte a fost revizuită și actualizată de Siewiorek, Bell și Newell (1982) sub titlul „*Computer Structures : Principles and Examples*". Aceste două cărți tratează în special arhitectura calculatoarelor seriale de mare succes (de exemplu, PDP 8 și 11, IBM 360), dar ultima include și o descriere a unor sisteme paralele (ILLIAC IV, STARAN, C. mmp., CRAY-1, TIASC). Alte mașini (ATLAS, IBM 370, UNIVAC 1100, DEC 10, CRAY 1) sînt descrise în numere speciale ale revistei Com. of the Ass. of Computing Machinery., dedicate arhitecturii calculatoarelor (ACM 1978) și în cartea lui Ibbett (1982), „*The Architecture of High Performance Computers*", care discută de asemenea CDC 6600 și

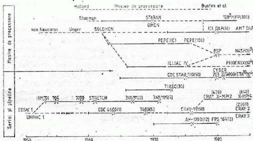


Fig. 1.2. Arbore de evoluție ce arată conexiunile arhitecturale și influențele marcante în cursul dezvoltării calculatoarelor paralele începînd din anii 50 și încheind la mijlocul anilor'80. Numerele din paranteză reprezintă numai un ghid asupra performanțelor în Mflop/s a calculatoarelor. Liniile întrerupte indică relațiile arhitecturale puternice.

7600, IBM 360 modelele 91 şi 195, TIASC şi CDC CYBER 205. Treceri în revistă ale arhitecturilor celor mai multor calculatoare comerciale se pot găsi într-o serie de rapoarte asupra tehnologiei calculatoarelor publicate de Auerbach (1976a). Recent, Zakharov (1984) a realizat o trecere în revistă a paralelismului şi procesării masivelor, incluzînd şi exemple de programare.

1.1.1. Paralelismul pînă în 1960

Se consideră că prima referinţă la paralelism în proiectarea calculatoarelor se găseşte în lucrarea generalului L.F. Menabrea, publicată în Biblioteque Universelle de Genève, în octombrie 1842, *„Sketch of the Analytical Engine Invented by Charles Babbage"* (vezi Morrison şi Morrison 1961, p. 244, Kuck 1977). El scrie, enunţînd utilitatea maşinii analitice :

„În al doilea rînd, economia de timp : pentru a ne convinge, trebuie să ne amintim că înmulţirea a două numere, fiecare cu 20 cifre, durează cel mult 3 minute. În acelaşi mod, cînd se execută un şir lung de operaţii identice, ca de exemplu : la formarea tabelelor matematice, maşina poate furniza mai multe rezultate în acelaşi timp, ceea ce va scurta mult timpul de execuţie".

Se pare că posibilitatea de a realiza operaţii în paralel nu a fost inclusă în proiectul final al maşinii de calculat a lui *Babbage*; totuşi, este clar că ideea utilizării paralelismului pentru a creşte performanţa maşinii i-a venit lui *Babbage* cu 100 de ani înainte ca tehnologia să facă posibilă implementarea ei. Ch. Babbage este autorul şi al altor idei fundamentale asupra calculului. În lucrările sale privind diferenţa mecanică şi maşinile analitice necesare pentru producerea unor tabele astronomice sigure (*Babbage 1822, 1864, 1910, Randell 1975*).

Primul calculator electronic numeric de uz general, ENIAC, a fost o maşină cu un grad înalt de paralelism şi descentralizare. Deşi a fost conceput şi proiectat în principal de *J.W. Mauchly şi J.P. Eckert Jr.*, ENIAC a fost descris mai mult de alţii (*Goldstine şi Goldstine 1946, Hartree 1946, 1950, Burks şi Burks 1981*). ENIAC poate fi considerat ca primul calculator MIMD, deoarece avea 25 unităţi de calcul independente (20 acumulatoare, 1 multiplicator, 1 divizor/extractor de radical şi 3 unităţi cu tabele memorate), fiecare urmînd propria secvenţă de operaţii şi cooperînd la rezolvarea unei singure probleme. Mai mult, el folosea intern aritmetica zecimală şi efectua calculele în paralel cu cele 10 cifre zecimale ale unui număr. ENIAC a fost proiectat şi construit la *Moore School of Engineering*. Universitatea din Pennsylvania, între 1943 şi 1946, în cadrul unui contract cu armata S.U.A., apoi a fost mutat la Aberdeen Ballistic Research Laboratory, unde a fost folosit pînă în anul 1955 pentru calculul traiectoriilor proiectilelor.

Arhitectura generală a lui ENIAC a fost concepută ca o versiune electronică a analizorilor diferenţiali mecanici care erau, la acea dată, cele mai avansate maşini pentru rezolvarea ecuaţiilor diferenţiale (*Bush 1931, 1936, Hartree 1950*). Aceste maşini foloseau o roată verticală în contact cu un disc orizontal pentru a forma un integrator mecanic. Dacă y este distanţa roţii la centrul discului, iar discul se roteşte cu un unghi dx, atunci roata se roteşte cu un unghi ydx. De aici, dacă y variază proporţional cu integrantul, rotaţia totală acumulată a roţii este ydx. Pentru a obţine o

soluție, s-au conectat prin axe mai mulți integratori de acest tip, conform tipului de ecuație diferențială ce trebuie rezolvată (vezi *Hartree 1950*). Astfel, analizorul diferențial mecanic era un dispozitiv analogic la care rotația unghiulară a axului era proporțională cu valorile variabilelor din problemă. În mod similar calculatoarelor analogice electronice de mai tîrziu, toate integratoarele (denumite apoi amplificatoare) lucrau simultan și în paralel. Acest mod de lucru în paralel a fost menținut de ENIAC. Totuși, deoarece trebuiau folosite diferențe finite, pentru a reprezenta coeficienții diferențiali, iar integralele au devenit sume finite, integratoarele analizorului diferențial au devenit unități acumulator (de exemplu, sumatoare) în ENIAC.

Folosind terminologia de azi, se poate descrie programarea lui ENIAC în felul următor : se atribuie diferitelor acumulatoare variabilele din formularea cu diferențe finite a problemei. Ieșirile și intrările acumulatoarelor se conectează împreună, așa cum cer ecuațiile, prin intermediul unor conectori externi; se inițializează pentru fiecare acumulator secvența de operații ce trebuie executată cu niște comutatoare (se poate stabili un microprogram diferit pentru fiecare din cele 20 acumulatoare), iar în unitatea centrală se programează în același mod programul ce inițiază execuția microprogramelor fiecărui acumulator la momentul de timp potrivit și sincronizează funcționarea lor. Se observă că nu există conceptul unui singur program memorat care să descrie întregul algoritm, și care ar fi urmat să fie executat de o arhitectură fixă de calculator — această idee importantă a calculatorului cu program memorat a fost implementată pentru prima dată la următoarea generație de calculatoare.

Prin contrast, arhitectura lui ENIAC se reorganiza pentru fiecare problemă, prin folosirea unor comutatori externi ce stabileau conexiunile între unități. Se poate spune că algoritmul era cablat în calculator. Este interesant că astfel de idei încep să sune foarte „moderne" din nou în anii '80, în contextul sistemelor MIMD, al masivelor VLSI reconfigurabile și al calculatoarelor specializate ce execută foarte rapid un set limitat de algoritmi implementați intern. Oricum, în 1940 nu era timpul potrivit pentru acest tip de arhitectură paralelă, așa cum se vede și din următorul citat din Burks (1981), membru al echipei de proiectare ENIAC :

„Paralelismul lui ENIAC a avut o viață scurtă. Mașina a fost terminată în 1946, an în care se proiectau deja calculatoare cu program memorat. S-a înțeles, mai tîrziu, că ENIAC putea fi reorganizat în maniera centralizată a acelor calculatoare noi iar atunci cînd s-a realizat aceasta era *mult mai ușor de formulat problema pentru mașină*. Această modificare a fost încheiată în 1948 . . .Ulterior nu s-a mai modificat conectica externă, iar mașina era programată prin comutatoare de la un punct central. În acest mod, primul calculator electronic de uz general, construit cu o arhitectură paralelă descentralizată, *a lucrat perioada cea mai îndelungată a existenței sale ca un calculator serial centralizat!"*

Dificultatea programării calculatoarelor paralele este o temă ce revine și urmează să se vadă dacă al doilea val al calculatoarelor paralele din anii '80 va avea mai mult succes decît primul!

Primele calculatoare cu program memorat — EDSAC la *Universitatea Cambridge* (*Wilkes și Renwick 1949*) și EDVAC la *Universitatea Pennsylvania* — ca și variantele lor comerciale (UNIVAC 1 : *vezi Eckert et al. 1951*) foloseau memorii cu linii de întîrziere cu mercur și o capacitate de 100 cuvinte. Deoarece, într-o astfel de memorie, biții succesivi ai unui număr sînt citiți secvențial începînd cu bitul cel mai puțin semnificativ

era natural şi economic ca operaţiile să se execute cu 2 operanzi, bit cu bit. Astfel, adunarea a două numere de 32 de biţi se realizează în 32 de cicli de maşină. Adunarea realizată astfel se numeşte *„serială, bit cu bit"*, iar prima utilizare a paralelismului în calculatoare s-a obţinut prin efectuarea operaţiilor cu toţi biţii unui număr în paralel.

Este interesant de menţionat că organizarea serială a fost considerată ca un avantaj introdus de componentele electronice în calculatoare, care realizau în mod tradiţional operaţiile în paralel. De exemplu, *Babbage* a respins principiul funcţionării seriale datorită timpului de execuţie mare (*Babbage 1864, Morrison şi Morrison 1961 pagina 34*), iar maşina sa realiza operaţii aritmetice pe 50 de cifre zecimale în paralel. Un calculator de birou obişnuit al anului 1940 lucra cu aproximativ 12 cifre zecimale în paralel. Plusul de viteză al componentelor electronice faţă de cele mecanice şi electromecanice a permis o creştere a performanţei între 1000 şi 10000, împreună cu o economie de echipament determinată de prelucrarea unei singure cifre la un moment dat. Acelaşi lucru este adevărat dacă se compară aritmetica paralelă a lui ENIAC, o maşină electronică anterioară, cu aritmetica serială a unor maşini ca EDVAC (*von Neumann 1945*). Tehnica circuitelor a progresat pînă la punctul unde intervalul de timp între impulsuri era la EDVAC de 1 microsec. În comparaţie cu 10 microsec. la ENIAC, ceea ce determină execuţia unei adunări seriale pe 32 biţi în 32 microsec., în comparaţie cu o operaţie zecimală cu 10 cifre executată în 200 microsec, la ENIAC. De asemenea, înmulţirea, prin sumare repetată, era mai rapidă la EDVAC decît la ENIAC, deşi ENIAC putea înmulţi 10 perechi de cifre zecimale în paralel.

Un calculator din prima generaţie care cuprindea anumite caracteristici de paralelism a fost ACE şi varianta sa comercială, English Electric DEUCE. Proiectarea modelului pilot ACE (Wilkinson 1953) a fost realizată în timpul prezenţei Dr. H.D. Huskey la Laboratorul naţional de fizică al Marii Britanii în anul 1947, după ideile Dr. A.M. Turing (Turing 1959) şi a devenit operaţional în 1951. Această maşină avea 11 linii de întîrziere, fiecare cu o capacitate de 32 de cuvinte de 32 de biţi şi un timp de recirculare de 1 msec. Unitatea aritmetică funcţiona serial cu o frecvenţă a tactului de 1 microsec. Maşina includea un lector de cartele, un perforator de cartele, un multiplicator care opera în paralel cu restul sistemului şi un pachet de instrucţiuni (pe care le-am denumi azi instrucţiuni vectoriale) pentru un număr limitat de operaţii asupra tuturor celor 32 de numere dintr-o linie de întîrziere. De exemplu, CPU realiza conversia zecimal-binar sau binar-zecimal în timp ce funcţiona şi lectorul de cartele sau perforatorul de cartele. De asemenea, semnul unei înmulţiri se stabilea de sumator în 2 ms, timpul necesar efectuării înmulţirii. Acestea sînt exemple de paralelism funcţional. O instrucţiune putea fi menţinută activă (de fapt prin repetare) pentru 32 de perioade de ceas, permiţînd, de exemplu, execuţia unor operaţii asupra tuturor celor 32 de numere dintr-o linie de întîrziere. Această facilitate a fost frecvent folosită pentru calculul sumelor de control şi este unul din exemplele cele mai vechi de instrucţiuni vectoriale. În versiunea comercială, DEUCE (*Haley 1956*), apărută în 1954, memoria rapidă a fost mărită la 12 linii cu întîrziere şi extinsă cu un tambur magnetic de 8000 de cuvinte. Paralelismul era evident în timpul transferului de pe tambur, deoarece în acest timp se puteau executa calcule în restul ma-

şinii. Suplimentar, DEUCE era dotat cu un divizor în virgulă fixă implementat hard, care putea lucra în paralel cu restul maşinii, ca şi multiplicatorul. Oricum, el utiliza aceleaşi registre cu multiplicatorul. Se foloseau ca registre cu acces rapid 3 linii de întîrziere pentru un cuvînt, 3 pentru două cuvinte şi 2 pentru patru cuvinte. Sumatoarele aveau asociate două linii de întîrziere, una cu un cuvînt, şi una cu două cuvinte, folosite şi ca acumulatoare.

Memoriile RAM au permis extragerea în paralel a biţilor unui cuvînt, ceea ce a dus la proiectarea unităţilor aritmetice ce operează în paralel pe mai mulţi biţi. Prima maşină experimentală care a implementat aritmetica paralelă a fost terminată în 1952 la Institutul de studii avansate (IAS) şi a fost urmată în 1953 de primul calculator comercial cu aritmetică paralelă, IBM 701. Ambele maşini foloseau ca memorii tuburi catodice electrostatice, inventate de *Williams şi Kilburn* (*1949*), ce au fost înlocuite după cîţiva ani de memorii cu ferite. Cel mai mare succes l-a avut IBM 704, din care s-au vîndut aproximativ 150 bucăţi. Această maşină avea nu numai aritmetica paralelă, dar şi prima unitate aritmetică în virgulă mobilă implementată hard, ceea ce i-a asigurat un plus de viteză faţă de maşinile ce simulau această unitate prin soft. Primul IBM 704 a fost vîndut în 1955, iar ultimul dezafectat în 1975. Istoria remarcabilă a acestei maşini din prima generaţie, operaţională mai bine de 20 de ani, este descrisă de Mc Laughlin (1975).

La IBM 704, ca şi la alte maşini ale timpului ei, informaţiile citite de echipamentele de intrare sau scrise la echipamentul de ieşire treceau printr-un registru al unităţii aritmetice, ceea ce împiedica funcţionarea simultană a unităţii aritmetice cu unităţile de I/E. Iniţial, echipamentul cuprindea un cititor de cartele (150—250 cartele/min), perforator de cartele (100 cartele/min) şi o imprimantă (150 linii/min). Curînd acestea au fost înlocuite de primele unităţi de bandă magnetică (cu o viteză de citire/scriere de 15000 caractere/secundă), de cel puţin 100 de ori mai rapide decît lectorul de cartele sau imprimanta. Un calculator specializat suplimentar, IBM 1401, asigură transferul offline între lectorul de cartele şi banda magnetică sau între banda magnetică şi imprimantă. Oricum viteza acestor unităţi, era de aproximativ 1000 de ori mai mică decît a unităţii centrale, de unde situaţiile de strangulare produse de canalele de I/E.

Problema I/E a fost parţial rezolvată prin funcţionarea în paralel cu imprimanta şi lectorul de cartele a unităţii aritmetice şi logice. A fost introdus astfel un calculator suplimentar, denumit *canalul de I/E*, a cărui funcţie era asigurarea transferului între echipamentele periferice lente, ca cititorul de cartele, benzile magnetice sau imprimanta şi memoria operativă a calculatorului. Odată iniţiat de unitatea centrală, transferul blocurilor mari de date continuă sub controlul canalului de I/E, în timp ce unitatea aritmetică funcţiona în paralel. Canalul de I/E are setul său propriu de instrucţiuni, specializat pentru operaţiile de I/E şi unitatea sa proprie de execuţie a instrucţiunilor împreună cu registrele asociate. În 1958 s-au adăugat 6 astfel de canale la IBM 704, rebotezat. IBM 709. Acesta este un prim exemplu de multiprocesare. IBM 709 a fost reproiectat în tehnologie tranzistorizată şi vîndut în 1959 ca IBM 7090. Această maşină, împreună cu versiunile ulterioare (IBM 7094 şi 7094 II) a avut un mare succes comercial, aproximativ 400 bucăţi fiind produse şi vîndute.

În faza inițială de proiectare a unui dispozitiv nou apar multe idei și variante, după care urmează o perioadă dominată de investiții masive pentru varianta adoptată. După aceea este dificil de a introduce noi inovații, în special datorită sumelor investite. Un exemplu este cel al motorului de automobil, care a inclus o mare varietate de principii, dar mai ales punerea la punct a motorului cu combustie internă. Cu greu și-ar propune cineva azi să înlocuiască acest tip de motor. Similar, în anii '50 s-au analizat multe proiecte de arhitecturi noi, deși, pînă spre 1980, numai calculatoarele SISD au avut un succes comercial cert. Oricum, se pare că tehnologia VLSI și microprocesoarele ieftine vor permite realizarea unora din aceste arhitecturi în anii '80.

În anul 1952 *Leondes și Rubinoff* au descris un calculator multi-operații, ce folosea o memorie cu tambur. Mașina DINA era un analizor digital pentru rezolvarea ecuațiilor lui Laplace. Un concept oarecum similar a fost introdus de Zuse (1958) în proiectarea unei ,,mașini de calcul a cîmpului". Von Neumann (1952) a stabilit principiile conectării spațiale a masivelor de procesoare, arătînd că un masiv bidimensional de elemente de calcul cu 29 de stări poate realiza toate operațiile, deoarece simulează comportarea unei mașini Turing. Acest rezultat teoretic a fost urmat de propunerile pentru realizări practice ale lui Unger (1958), care poate fi considerat părintele calculatoarelor anilor '70, SOLOMON, ILLIAC IV și ICL DAP.

În mod similar, lucrările lui Holland (1959), ce descriau un ansamblu de procesoare executînd fiecare propriul flux de instrucțiuni, pot fi considerate primele proiecte de sistem multiprocesor, de la care pleacă proiectele ulterioare de microprocesoare interconectate, ca acela propus 20 de ani mai tîrziu de Pease (1977) și de Bustos et al (1979) pentru rezolvarea problemelor de difuzie. Pease a propus un proiect de cub binar n-dimensional format din 2^n microprocesoare conectate ca un cub 1—, 2— sau n— dimensional, adecvat pentru rezolvarea unei game largi de probleme ca, de exemplu, efectuarea transformatei Fourier. Se puteau folosi pînă la 16384 microprocesoare, ($n = 14$). Hipercubul anunțat de IMS Associates (Millard 1975) se baza pe ideile lui Pease : era format dintr-un cub 4-dimensional cu 2 microprocesoare în fiecare nod, unul pentru comunicații și celălalt pentru operații. Acest hipercub particular nu reprezintă nimic acum, totuși ideea a reapărut la începutul anilor '80 sub forma proiectelor Cosmic Cube și Intel Personal Supercomputer (vezi §1.1.8).

1.1.2. Calculatoare scalare rapide

Un calculator scalar execută instrucțiuni care se referă numai la operanzi formați dintr-un singur număr, spre deosebire de calculatoarele vectoriale care posedă și instrucțiuni pentru manipularea unui set ordonat de numere (vector). Istoria dezvoltării calculatoarelor scalare în anii '60 și '70 este istoria introducerii din ce în ce mai mult a paralelismului în proiectarea generală a calculatoarelor SISD, ca IBM 7090, care a devenit un model standard pe la sfîrșitul anilor '50.

Calculatorul ATLAS (*Kilburn et al 1962, Sumner et al 1962, Howarth et al 1961, 1962, Lavington 1978*) a avut o influență profundă atît asupra

arhitecturii cît şi a soft-ului. Această maşină a fost concepută la Universitatea Manchester în jurul anului 1956 şi proiectată de un colectiv mixt universitate-industrie sub conducerea prof. Kilburn. Prototipul a devenit operaţional în 1961, iar primul exemplar comercial a fost produs de firma *Ferranti Ltd* (mai tîrziu componentă a ICL) în 1963. ATLAS a devenit cunoscut prin introducerea unui sistem de operare cu facilităţi de multiprogramare complexă, cu o memorie virtuală de mare capacitate şi un sistem de întreruperi. Sistemul de operare organiza alocarea de resurse programelor în diferite etape ale execuţiei. Spaţiul de adresare virtuală era văzut de utilizator ca o memorie cu un singur nivel de aproximativ 10^6 cuvinte, translatat în mai multe nivele fizice ce cuprindeau memorii cu ferite (16 Kcuv). tambur magnetic (96 Kcuv.) şi benzi magnetice. Translatarea se realiza cu un sistem de paginare care transfera informaţii între diferitele nivele ale memoriei în unităţi de 512 cuvinte, denumite pagini. Sistemul de întreruperi permitea lucrul independent al dispozitivelor de I/E lente şi întreruperea CPU, numai cînd era absolut necesar.

În plus faţă de cele prezentate, ATLAS utiliza paralelismul şi pentru creşterea vitezei de calcul. Memoria cu ferite, cu un ciclu de 2 microsec. era împărţită în 4 blocuri independente (denumite stive) care făceau posibilă, în circumstanţe favorabile, regăsirea într-un ciclu de memorie a două instrucţiuni şi a celor doi operanzi simultan. Unitatea de calcul lucra paralel pe bit (Kilburn et al. 1960). Paralelismul funcţional era prezent sub forma unui sumator separat de 24 pe biţi, pentru calculul indexului, denumit unitatea aritmetică B, în plus faţă de unitatea aritmetică în virgulă fixă şi mobilă pe 48 de biţi (40 biţi mantisa şi 8 biţi exponentul), ce folosea un singur acumulator. Unitatea aritmetică B lucra în asociaţie cu 128 registre index de 24 biţi (denumite liniile B), construite din ferite şi cu un timp de acces de 0,7 microsec. Instrucţiunile aveau un singur cîmp de adrese şi specificau o operaţie între conţinutul acelei adrese şi conţinutul acumulatorului principal. O instrucţiune de 48 biţi era formată dintr-un cîmp de de cod de 10 biţi, o adresă în memoria principală de 24 biţi şi două adrese de registru index de 7 biţi. Adresa operandului se obţinea prin sumarea conţinutului unuia sau al ambelor registre index la adresa principală. Principiul pipelining era utilizat pentru suprapunerea următoarelor faze ale execuţiei instrucţiunilor : extragerea instrucţiunilor, calcularea în unitatea B a adresei operandului, extragerea operandului şi execuţia calculelor în acumulatorul principal pe 48 biţi. Aplicarea efectivă a acestui principiu poate fi apreciată prin faptul că execuţia unor adunări în virgulă mobilă ar fi necesitat 6 microsec./operaţie dacă s-ar fi executat secvenţial toate fazele şi numai 1,6 microsec./operaţie în medie cu pipelining. În orice moment se puteau afla în execuţie între 2 şi 4 instrucţiuni. Timpul corespunzător pentru înmulţiri în virgulă mobilă era de 5 microsec. Logica lui ATLAS era implementată cu tranzistoare cu germaniu şi diode cu un timp de propagare/poartă mediu de 12 ns. Se utilizau aproximativ 80000 tranzistoare.

Pentru a folosi efectiv posibilităţile de execuţie paralelă oferite de unităţile aritmetice multiple, registre şi memorii, trebuie analizat fluxul instrucţiunilor şi stabilit care pot fi executate concurent, fără alterarea logicii programului. După detectarea paralelismului se impune execuţia optimă a instrucţiunilor. Aceste aspecte sînt tratate de Keller (1976) şi Kuck (1978)

şi sînt incluse în arhitectura calculatoarelor scalare cele mai rapide ca CDC 6600 şi IBM 360/91.

Introducerea graduală a paralelismului funcţional şi pipeline poate fi observată în evoluţia conceptului de calculator scalar la firma CDC, sub influenţa lui Seymour R. Cray (membru fondator al firmei în 1957, vice-preşedinte şi şef proiectant). CDC 6600, comercializat începînd cu 1964, a fost primul calculator ce utiliza paralelism funcţional ca o caracteristică esenţială a arhitecturii sale (*Thornton 1964*). *Thornton* (*1970*), responsabil pentru detaliile de proiectare, descrie foarte bine proiectarea maşinii în cartea sa „*Design of a Computer—The Control Data 6600*". CDC 6600 avea o memorie cu ferite cu un ciclu de 1 microsec., împărţită în 32 blocuri independente care lucrau în paralel; 10 unităţi funcţionale separate pentru înmulţire, împărţire, adunare etc., 10 procesoare periferice pentru gestiunea dispozitivelor de I/E lente. Procesoarele de I/E, deşi partajează aceeaşi unitate aritmetică şi de comandă, execută instrucţiuni diferite asupra datelor din blocuri de memorie separate. Acesta este un exemplu de sistem multiprocesor. Unitatea aritmetică funcţiona în virgulă mobilă (pentru cuvinte de 60 biţi), exceptînd unitatea incrementală (opera pe 18 biţi) folosită la calculul adreselor şi evidenţa buclelor. CDC 6600 a avut un astfel de succes încît a acaparat o mare parte din piaţa calculatoarelor ştiinţifice, anterior dominată de IBM 7090 şi 7094. În anul 1969 CDC a lansat o versiune îmbunătăţită a lui 6600, CDC 7600. Viteza noului calculator era de 4 ori mai mare deoarece perioada ceasului a fost redusă de la 100 ns pentru 6600 la 27,5 ns. Cele 10 unităţi funcţionale organizate serial la 6600 au fost înlocuite cu 8 unităţi funcţionale pipeline şi o unitate serială pentru împărţire. Datorită vitezei unităţilor pipeline nu a mai fost necesară duplicarea unităţilor de înmulţire şi incrementală. A fost adăugată o unitate funcţională suplimentară pentru numărarea biţilor de „1" într-un cuvînt. Arhitectura calculatoarelor CDC 6600 (rebotezat CYBER 70 model 74) şi CDC 7600 (rebotezat CYBER 70 model 76) a avut un mare succes pe piaţa calculatoarelor ştiinţifice. Au fost instalate peste 50 de exemplare din ultimul tip.

Şi istoria producţiei de calculatoare IBM în anii 60 şi '70 ilustrează o introducere graduală a paralelismului. În 1956 a fost abordat cu mare entuziasm un contract cu Laboratorul ştiinţific Los Alamos pentru un calculator de 100 de ori mai rapid decît IBM 704. Acesta a fost botezat STRETCH (Dunwell 1956, Bloch 1959) şi a fost comercializat mai apoi pentru o perioadă scurtă de timp ca IBM 7030. Noutăţile introduse constau în extragerea, decodificarea, calculul adreselor şi extragerea operanzilor mai multor instrucţiuni în avans ca şi împărţirea memoriei în două blocuri independente care puteau alimenta unitatea aritmetică în paralel. Astfel, viteza de transfer a informaţiei către şi de la memorie era multiplicată de un factor egal cu numărul blocurilor de memorie. Acesta este primul exemplu de aplicare a paralelismului la memorie şi a permis o adaptare mai bună între memoria cu ferite şi procesor. Aproape toate calculatoarele mari au folosit ulterior împărţirea memoriei în blocuri. Primul STRETCH a fost livrat în 1961 dar nu a atins performanţele propuse. Deoarece şi aspectele financiare au nemulţumit conducerea companiei, după instalarea a 7 exemplare, construcţia sa a fost sistată. În continuare firma a oferit calculatoarele mai lente dar foarte populare ale seriei 7090.

Se părea că după experienţa cu STRETCH, IBM şi-a pierdut interesul pentru calculatoarele de mare viteză. În 1964 a fost anunţată seria IBM 360, dar care nu conţinea nici un model cu performanţe similare lui CDC 6600, comercializat incepînd cu acel an.

Succesul dramatic al înlocuirii calculatorului IBM 7090 cu CDC 6600 în multe centre ştiinţifice a obligat IBM la un răspuns. Astfel, în anul 1967 a fost lansat IBM 360/91 (Anderson et al. 1967) cu performanţe duble faţă de CDC 6600. Maşina avea facilităţile lui STRETCH dar, ca şi CDC 6600, unităţile de calcul în virgulă mobilă şi de calcul al adresei erau separate şi pipeline. Principiul pipeline a fost folosit şi pentru creşterea vitezei de execuţie a instrucţiunilor, operaţiile de extragere a instrucţiunilor, decodare, calculul adresei, extragerea operanzilor fiind suprapuse pentru instrucţiuni succesive. Astfel, mai multe instrucţiuni se găseau simultan în faze diferite ale execuţiei lor şi circulau prin pipeline. În 1969 a fost lansat CDC 7600 care a depăşit de două ori performanţele lui 360/91. IBM a răspuns prin lansarea în 1971 a modelului 360/195, cu performanţe comparabile cu ale lui CDC 7600. IBM 360/195 (Murphy şi Wade 1970) combina arhitectura lui 360/91 cu memoria cache introdusă la 360/85. Ideea introducerii unei memorii tampon foarte rapidă (denumită şi cache) între memoria principală şi registrele unităţii aritmetice se regăseşte la calculatorul ATLAS (Fotheringham 1961). Cache-ul sistemului 360/85 cu o capacitate de 32768 cuvinte şi un timp de acces de 162 ns, stoca informaţiile folosite cel mai des în blocuri de 64 octeţi. Dacă informaţia solicitată de o instrucţiune nu se găsea în cache, blocul care o conţinea din memoria principală (4 MB împărţiţi în 16 blocuri) înlocuia blocul cel mai puţin folosit din cache. S-a observat că pentru calcule numeroase referinţele la memorie tind să se concentreze în jurul unei regiuni limitate de adrese. În acest caz cele mai multe referinţe vor fi la memoria cache, iar performanţele memoriei lente de 4 MB vor fi în mod efectiv cele ale memoriei cache rapide.

Gene Amdahl, care a fost arhitectul şef al seriei IBM 360 (Amdahl et al. 1964) a fondat o companie separată în 1970 (Amdahl Corp) pentru a produce o gamă de calculatoare compatibilă cu IBM 360. Aceste maşini au marcat un pas important în evoluţia tehnologiei calculatoarelor şi a paralelismului. Prima maşină a seriei ,AMDAHL 470V/6, folosea pentru prima dată tehnologie LSI pentru circuitele logice ale CPU (ECL cu 100 porţi/cip), de unde denumirea de maşină din generaţia a patra. În 1975 s-au produs 6 maşini din care una pentru NASA şi una pentru Universitatea Michigan. Utilizarea LSI a dus la reducerea la aproximativ 1/3 a maşinii în comparaţie cu IBM 360/168. Deşi unităţile aritmetice nu erau pipeline se atingea o viteză mare de calcul prin aplicarea principiilor pipeline la procesarea instrucţiunilor. Execuţia instrucţiunilor era împărţită în 12 suboperaţii care foloseau 10 circuite separate. La fiecare două perioade de ceas (64 ns) intra în execuţie o nouă instrucţiune şi, astfel, se aflau în execuţie simultană 6 instrucţiuni, fiecare într-o fază diferită (pipeline). O memorie bipolară tampon (cache) de mare viteză (16 kB şi timp de acces de 65 ns) a crescut viteza de acces la informaţii, pentru o memorie principală de 8 MB şi un timp de acces de 650 ns.

1.1.3. Calculatoare vectoriale pipeline

În anul 1972 Seymour Cray a părăsit CDC pentru a înființa propria sa companie, Cray Research Inc., cu scopul de a realiza cel mai rapid calculator din lume. În timpul extraordinar de scurt de numai 4 ani a fost proiectat și construit CRAY-1 primul exemplar fiind livrat Laboratorului științific Los Alamos în 1976. CRAY-1 (Russell 1978, Dungworth 1979) urmează linia de evoluție a calculatoarelor 6600 și 7600. Are 12 unități funcționale, toate pipeline, o perioadă a ceasului de 12,5 ns și o memorie cu 16 blocuri a cîte un milion de cuvinte (ciclu de 50 ns). Principala inovație o constituiau cele 8 registre vectoriale, fiecare capabil de a memora 64 numere în virgulă mobilă de 64 biți, ca și un set de 32 instrucțiuni pentru manipularea și efectuarea de operații cu vectori. Trei unități funcționale erau rezervate pentru operații vectoriale, iar trei partajate de instrucțiuni scalare. CRAY-1 a fost denumit calculator vectorial, deoarece poseda instrucțiuni pentru operații cu vectori. A fost primul calculator vectorial pipeline cu succes comercial și rămîne de departe cu cel mai mare succes; pînă la sfîrșitul anului 1984 s-au vîndut 88 CRAY-1 și CRAY X-MP. CRAY-1 atinge în mod obișnuit viteze de 130 Mflop/s (milioane de operații în virgulă mobilă pe secundă) pentru probleme adecvate (de exemplu, înmulțirea matricelor).

Principalele neajunsuri ale proiectului CRAY-1 original au fost (a) posibilități de I/E inadecvate , (b) viteza de transfer mică între memoria operativă și registrele vectoriale (lărgimea de bandă mică) și (c) absența unei instrucțiuni hardware de dispersare/grupare. Acestea au determinat performanțe, în mod semnificativ mai mici decît cele teoretic posibile, pentru o serie de probleme reale. CRAY-1S, anunțat în 1979, a rezolvat primul neajuns prin introducerea unui Subsistem de I/E care elibera CPU de gestiunea operațiilor de I/E și a zonelor de memorie tampon, asociat calculatorului front-end, discurilor și altor dispozitive periferice. Subsistemul este compus din 2 pînă la 4 procesoare de I/E și o memorie tampon de 1/2 sau 1 MB. Subsistemul de I/E are un canal direct la memoria operativă ce lucrează cu o viteză de transfer de 850 mil. biți pe secundă. De asemenea, CRAY-1S are memoria principală mai mare, de 4 Mcuv. în loc de 1Mcuv., în același spațiu fizic, obținută prin înlocuirea cipurilor de 1 Kb cu cele de 4 Kb.

Următorul pas important în evoluția seriei de calculatoare CRAY-1 a fost anunțul lui CRAY X-MP în 1982. Acesta este un proiect multiprocesor ce poate fi descris ca 2 calculatoare CRAY-1 cu o memorie comună de 2 sau 4 Mcuv. Fiecare CPU are 4 porturi în memorie, unul pentru I/E și 3 pentru utilizare de către registrele vectoriale, crescînd astfel lărgimea de bandă maximă pentru transferul datelor de 8 ori, ceea ce rezolvă problema lărgimii de bandă mici (b). În absența conflictelor de acces la un bloc de memorie, două fluxuri vectoriale de intrare și un flux de ieșire pot accesa acum memoria simultan în ambele CPU și fiecare poate alimenta un pipeline aritmetic la viteza lui maximă. Cele 2 CPU pot lucra cu părți diferite ale problemei sau cu probleme independente (job-uri). Sincronizarea se realizează fie prin locații ale memoriei operative, fie printr-un set de registre speciale. Un nivel înalt de integrare (16 porți/cip, în comparație cu 4 sau 5 porți/cip la CRAY-1) permite celor 2 CPU să încapă în același spațiu fizic ca și CRAY-1. Memoriile secundare de mare viteză sînt dis-

pozitive SSD (solid state device), formate din pînă la 32 Mcuv. în 64 blocuri de memorie MOS volatilă. Viteza de transfer între SSD și memoria operativă este de 1250 MB/s. Astfel, versiunea inițială a lui CRAY X-MP a eliminat primele două inconveniente ale lui CRAY-1. În anul 1984 s-a anunțat un model cu 4 CPU care elimină și ultima problemă, prin includerea unei instrucțiuni hardware corespunzătoare. Cel mai avansat în gama sa, CRAY-X-MP, atinge o performanță maximă de 840 Mflop/s și are o memorie operativă bipolară ECL de 8 Mcuv. (64 biți) organizată în 64 blocuri cu acces paralel. CRAY X-MP este descris în detaliu în capitolul 2 (§2.2).

În 1981 Seymore Cray a demisionat din postul de președinte al firmei Cray Research pentru a-și dedica întregul timp dezvoltării calculatorului CRAY-2 și a altor proiecte de viitor. Noua tehnologie de imersiune în lichid, folosită la CRAY-2, a fost experimentată în 1981 (Cray 1981). Ea constă în scufundarea întregii mașini într-o baie de lichid inert, fluorocarbon, un lichid de răcire. Cînd carcasa este un container transparent — creîndu-se așa-numitul calculator „globul cu peștele de aur" — se pot observa efecte foarte frumoase datorită turbulenței din lichid. Eficiența sporită a răcirii permite utilizarea unei densități mai mari de componente și un CPU mai mic. Întreaga mașină ocupă un cilindru de 1,2 m înălțime și 1,4 m în diametru. Plăcile cu circuite sînt plasate după cele 3 dimensiuni, ceea ce determină o lungime maximă a firelor de 16 inch, în comparație cu plasarea bi-dimensională și lungimea firelor de 1,2 m la CRAY-1.CRAY-2 este proiectat să aibă pînă la 4 CPU și o perioadă a ceasului de 4ns. Dacă cele două unități pipeline în virgulă mobilă (+, ×) ale fiecărui CPU lucrează simultan, atunci la fiecare nanosecundă se termină 2 operații aritmetice, ceea ce înseamnă o viteză de 2 Gflop/s. La fel ca la CRAY X-MP, noi CPU vor putea fi adăugate la mașinile ulterioare. Fiecare CPU are o arhitectură similară lui CRAY-1, dar includerea unei memorii locale de 16 kcuv. pentru rezultate intermediare face mașina incompatibilă cu seria CRAY-1 la nivelul codului mașină. CRAY-2 va folosi sistemul de operare UNIX, iar adresele pe 32 biți asigură un spațiu de adresare total de 256 Mcuv. Mașini CRAY-2 prototip, cu 1 CPU și memorie de 16 Mcuv. au fost livrate la NMFECC Livermore și Universitatea Minnesota în 1984/5, iar trei mașini cu 4 CPU au fost vîndute în 1985, inclusiv una pentru NASA Ames Research Laboratory, cu 256 Mcuv. În anul 1986 s-au realizat 4 instalări cu 4 CPU și fie 64, fie 256 Mcuv. de memorie. În Europa a fost comandat primul CRAY-2 în 1985 pentru landul Baden-Wurtemberg, urmînd să fie instalat la Universitatea din Stuttgart. În 1987 au mai fost instalate CRAY-2 la Ecole Polytechnique din Paris, și la UKAEA Harwell Laboratory în Marea Britanie. Inițial, CRAY-2 va folosi aceleași circuite logice cu 16 porți ca și CRAY X-MP. Probabil, următorul pas va fi folosirea circuitelor pe bază de arseniură de galiu, care ar permite reducerea perioadei ceasului la aproximativ 1 ns. Această mașină, planificată pentru 1990, se va numi, probabil, CRAY-3. Va avea 16 procesoare, o memorie comună de 1 Gcuv. și o performanță maximă de 15 Gflop/s. CRAY 2 este descris în detaliu în §2.2.7.

Alte două calculatoare vectoriale pipeline CDC-STAR 100 și TIASC (Theis 1974, Hockney 1977) au origini mai vechi decît CRAY-1 .STAR 100 (Hintz și Tate 1972) a fost conceput prin 1964 ca procesor vectorial

cu un set de instrucţiuni bazat pe limbajul APL (Inverson 1962). Ar fi trebuit să realizeze 100 Mflop/s pentru vectori de dimensiuni mari care interveneau în multe probleme ştiinţifice ale Laboratorului Lawrence Livermore (LLL). În 1967 a început proiectarea. După o perioadă de gestare de 6 ani a devenit operaţională prima maşină, iar în anii '74—'75 au fost livrate două bucăţi către LLL. Existau însă două dezavantaje majore: unul tehnologic şi unul care ţinea de proiectul propriu-zis, care au dus la dezvoltarea lui lentă. Iar cînd a devenit disponibil, STAR 100 a suferit datorită proiectării sale la începutul anilor '60. Memoria cu ferite cu un ciclu de 1,2 microsec. a fost depăşită de memoriile cu semiconductori, iar perioada ceasului de 80 ns este lentă în comparaţie cu maşinile similare disponibile la mijlocul anilor '70. Mai mult, cele mai bune calculatoare organizate serial, ca CDC 7600 şi IBM 360/195, aveau unităţile aritmetice scalare mai rapide. STAR 100 depăşeşte aceste calculatoare doar pentru operaţii cu vectori de sute sau mii de componente. În consecinţă, s-au vîndut numai 4 maşini (2 pentru LLL, una pentru NASA) sau au fost reţinute de companie ca o facilitate pentru reţeaua de comunicaţii CYBERNET. Nici un STAR 100 nu a fost vîndut unui beneficiar obişnuit. Apoi, STAR 100 a fost reproiectat în tehnologie LSI cu memorii semiconductoare şi oferit pe piaţă în 1979 ca CYBER 203E (Kascic 1979). Această maşină „rebotezată" CYBER 205, este un adversar serios pentru CRAY-1. De aceea o vom considera ca al doilea exemplu de arhitectură pipeline şi o vom trata în §2.3. CYBER 205 diferă de CRAY-1 în procesarea instrucţiunilor vectoriale, deoarece nu are registre vectoriale, conţine mai multe pipeline de uz general, spre deosebire de cele specializate ale lui CRAY-1. Primul sistem a fost livrat Oficiului meteorologic al Marii Britanii în 1981.

În septembrie 1983 CDC a fondat o companie separată, ETA Systems Inc., pentru a accelera dezvoltarea ulterioară a supercalculatoarelor sale. Control Data a păstrat 40% din proprietatea firmei ETA Systems, care a anunţat ca obiectiv realizarea unui calculator cu viteză de 10 Gflop/s, ETA[10]. Păstrînd arhitectura lui CYBER 205, acest calculator este de 3 pînă la 5 ori mai rapid ca acesta, graţie celor 8 procesoare. Astfel, compatibilitatea cu întregul soft dezvoltat pentru CYBER 205 se păstrează. Viteza suplimentară şi nivelul mai mare de integrare se obţin datorită folosirii tehnologiei CMOS răcită la temperatura azotului lichid (—200°C). Un cip CMOS produs de Honeywell, cu pînă la 20000 porţi disipă 2 waţi, în comparaţie cu cei 4 waţi disipaţi de circuitele ECL cu 250 porţi folosite la CYBER 205. Astfel, unitatea centrală va ocupa numai 1,5 m pătraţi. ETA[10] este descris în detaliu în §2.3.7.

Proiectul TIASC (Watson 1972, Ibbett 1982) a demarat în 1966 cu scopul de a realiza un sistem pentru rezolvarea problemelor de analiză a datelor seismice, implicînd utilizarea pe scară largă a transformatei Fourier. Proiectul se baza pe 1,2 sau 4 pipeline de uz general identice, fiecare capabil de a executa instrucţiuni elementare cu operanzi vectoriali. Instrucţiunile erau interpretate de una sau două unităţi procesoare. Acesta este un alt exemplu de sistem multiprocesor. Memoria cu semiconductori era formată din 8 blocuri cu un ciclu de 320 ns. Teoretic, la o funcţionare optimală a celor 4 unităţi pipeline, s-a obţinut o viteză de 50 Mflop/s. Numărul unităţilor aritmetice pipeline şi al celor de interpretare a ins-

trucțiunilor era opțional. Primul TIASC a devenit operațional în 1973, iar de atunci au mai fost construite încă 6 bucăți, cele mai multe folosite de companie sau de asociații ei. Cele mai importante vînzări au fost o configurație cu 4 unități pipeline pentru Laboratorul geofizic de dinamica fluidelor de la Princeton și o configurație cu 2 unități pipeline pentru Laboratorul de cercetări navale din Washington. După instalarea celor 6 sisteme, TI a sistat producția. Și acest calculator a suferit, ca și STAR 100, consecințele unor unități scalare lente în comparație cu CDC 7600.

Prima companie japoneză care s-a lansat în domeniul supercalculatoarelor a fost Fujitsu, care a anunțat în anul 1982 calculatoarele vectoriale pipeline FACOM VP-100 și VP-200 (Miura și Uchida 1984). În mod evident aceste mașini au fost proiectate pentru a combina caracteristicele cele mai avantajoase ale calculatoarelor CRAY-1 și CYBER 205. Ele posedă unități scalare și vectoriale separate, care pot lucra simultan, ca la CYBER 205; pe de altă parte, unitatea vectorială este compusă din registre ca la CRAY-1. Există unități pipeline separate pentru adunare/operații logice, înmulțire și împărțire, ce lucrează cu datele păstrate în memoria registrelor vectoriale (64KB). O caracteristică unică a acestor mașini este posibilitatea reconfigurării memoriei vectoriale în mod dinamic sub controlul programului. Se pot configura fie 8 registre vectoriale a cîte 1024 elemente, fie 256 registre a 32 elemente, fie prin orice factor de 2 între aceste limite. Între registrele vectoriale și memoria operativă există 2 unități pipeline pentru încărcare/memorare (în comparație cu 1 la CRAY-1 și 3 la CYBER 205 și CRAY X-MP). Capacitatea maximă a memoriei operative este de 256 Mcuv. (de 8 ori mai mult față de CYBER 205 sau CRAY-1). FACOM VP posedă instrucțiuni de mascare, similare celor de la CYBER 205, ca și o instrucțiune hardware de adresare vectorială indirectă (instrucțiuni de dispersare/grupare), prezentă la CYBER 205, dar absentă la CRAY-1. Unitatea vectorială are o perioadă a ceasului de 7,5 ns ce determină o viteză maximă de calcul de 533 Mflop/s la VP-200. Această viteză ca și capacitatea memoriei sînt înjumătățite la VP-100. Prima livrare a constat dintr-un VP-100 la Universitatea din Tokyo, la sfîrșitul anului 1983. În cursul aceluiași an au fost anunțate încă două calculatoare vectoriale pipeline japoneze, HITACHI S-810 model 10/model 20 (630 Mflop/s maximum) și NEC SX-1/SX-2 (maximum 1300 Mflop/s). Toate aceste calculatoare japoneze sînt descrise pe larg în capitolul 2, §2.4.

IBM a anunțat în anul 1985 (9 ani după livrarea primului CRAY-1) prima mașină pentru calculul vectorial, ca parte a seriei System/370. Aceasta este un multiprocesor IBM 3090 scalar, cu o performanță maximă de aproximativ 5 Mflop/s, la care pot fi adăugate opțional facilități vectoriale care cresc de aproximativ 3 sau 4 ori performanța scalară. Acest raport între viteza vectorială și cea scalară este mult mai mic decît cel furnizat de mașinile concurente, dar este considerat de IBM ca cel mai economic. Fiecare procesor al sistemului 3090 poate primi o singură facilitate vectorială, iar numărul maxim actual (1987) de procesoare este 6, care poate însă crește. Deci, IBM 3090-VF realizează atît prelucrări vectoriale (SIMD), cît și multitasking (MIMD) prin utilizarea procesoarelor în mod independent. Fiecare procesor 3090 are o memorie cache de 64 KB, folosită atît de unitatea scalară, cît și de cea vectorială pentru instrucțiuni

şi date. Modelul 400, cu 4 procesoare, are o memorie operativă de 128 MB cu posibilitatea extensiei la 256 BM. Fiecare facilitate vectorială are 16 registre vectoriale, fiecare cu capacitatea de 128 numere pe 64 biţi. Ciclul de 18,5 ns corespunde la o performanţă maximă de 54 Mflop/s, pe pipeline. Unitatea vectorială are unităţi pipeline independente pentru înmulţire şi adunare, ceea ce determină o performanţă teoretică maximă de 108 Mflop/s. Aceste valori pot fi însă atinse numai pentru un cod optimizat, care păstrează datele necesare în registrele vectoriale sau în memoria cache timpul cel mai îndelungat, minimizînd astfel timpul de acces la memoria centrală sau extinsă, care reprezintă o serioasă strangulare în sistem. De exemplu, viteza obţinută pentru o singură operaţie vectorială diadică în FORTRAN (vezi §1.3.3) cu toate datele în memoria cache este de 13 Mflop/s. Dacă datele se află în memoria operativă, această viteză se reduce la 7,3 Mflop/s pentru un pas între elementele succesive ale vectorului de 1 (în comparaţie cu 70 Mflop/s la CRAY X-MP, vezi §2.2.6) şi la 1,7 Mflop/s pentru un pas de 8.

1.1.4. Calculatorul Solomon şi ce a urmat după el

Un moment important în istoria paralelismului l-a reprezentat, fără îndoială apariţia lucrării lui Slotnick et al. (1962) intitulată „The SOLOMON Computer". Principiile acestui calculator au fost descrise şi de Gregory şi McReynolds (1963) şi constituie o aplicare a ideilor lui Unger (1958). Acronimul înseamnă „Simultaneous Operation Linked Ordinal MOdular Network" şi descrie un masiv bidimensional de 32×32 elemente procesoare, fiecare cu o memorie de 128 cuvinte a 32 biţi şi o unitate aritmetică ce lucrează serial pe bit sub controlul unei unităţi de comandă unice. Contrar evoluţiei calculatoarelor seriale spre calculatoare vectoriale pipeline conceptul calculatorului SOLOMON a însemnat o modificare radicală în arhitectura calculatoarelor, cu o influenţă substanţială asupra cercetării şi proiectării. Calculatorului SOLOMON nu a fost niciodată construit aşa cum a fost descris în lucrarea din 1962, dar este punctul de pornire atît pentru ILLIAC IV, Burroughs PEPE, cît şi a modelelor STARAN (Goodyear Aerospace) şi ICL DAP, denumite adeseori procesoare asociative. Vom prezenta aceste calculatoare separat.

Departamentul Apărării al S.U.A. a propus Universităţii Illinois în 1966 un contract pentru proiectarea unui calculator similar lui SOLOMON, care a devenit cunoscut mai tîrziu ca ILLIAC IV. Proiectul original este descris de Slotnick (1967) în altă lucrare de mare rezonanţă, „Unconventional Systems" şi de Barnes et al. (1968) şi McIntyre (1970). Iniţial, sistemul era format din 4 subsisteme, fiecare cu o unitate de comandă pentru execuţia unui flux unic de instrucţiuni de către 64 elemente procesoare în virgulă mobilă. Fiecare element procesor conţinea 2000 cuvinte de 64 biţi înregistraţi pe peliculă de film, iar în fiecare bloc elementele procesoare erau conectate într-un aranjament bidimensional 8×8. Cele 4 subsisteme (blocuri) trebuiau conectate cu un bus de I/E paralel şi cu acces la un disc magnetic de capacitate mare de unde se citeau job-urile şi unde se scriau rezultatele. Deoarece se considera că fiecare element procesor execută o operaţie în virgulă mobilă în 240 ns, s-ar fi putut obţine

o viteză de 1 Gflop/s. Deşi nu s-a construit decît un subsistem, iar viteza atinsă a fost de 50 Mflop/s, producerea lui ILLIAC IV a avut o influenţă considerabilă asupra tehnologiei calculatoarelor şi arhitecturii. Principiile de funcţionare ca şi unele aplicaţii sînt tratate în Feierbach şi Stevenson (1979a) şi Slotnick (1971).

Povestea construirii calculatorului ILLIAC IV este descrisă de Falk (1976) în articolul său „Reaching for the Gigaflop". Această maşină este prima care a folosit circuite de memorie cu semiconductori (256 porţi logice în tehnologie bipolară) pentru memoria centrală, după ce s-a ajuns la concluzia că nu este spaţiu suficient pentru memoria pe pelicula de film. S-a intenţionat să se realizeze 20 porţi logice pe circuit (MSI), dar nu s-au realizat decît 7 (SSI). Rezultatul s-a datorat alegerii tehnologiei ECL în locul TTL. De asemenea, la ILLIAC IV s-au introdus plăcile cu 15 straturi şi cu acest prilej şi metoda de conectare adecvată. Producerea lui ILLIAC IV, redus la un subsistem a fost cedată firmei Burroughs, iar calculatorul a fost livrat în 1972 Centrului de cercetări Ames al NASA. Dar, abia în 1975 s-au oferit servicii utilizabile şi aceasta pentru o perioadă a ceasului de 80 ns (în loc de 40 ns) cu încetinirea corespunzătoare a tuturor operaţiilor.

Dacă se ia în considerare faptul că preţul producerii lui ILLIAC IV a fost de 4 ori mai mare decît cel propus iniţial, iar performanţele de 10 ori mai mici, acesta a reprezentat un eşec. Totuşi, influenţa sa a fost profundă şi probabil în anii '80 vom vedea arhitecturi de calculatoare similare, mai ales că progresele tehnologice le fac posibile. Ca şi alte proiecte majore, ILLIAC IV a depăşit posibilităţile tehnologice ale timpului său. Pentru acest calculator s-a dezvoltat un soft considerabil, inclusiv 4 limbaje care pot exprima paralelismul calculatorului şi noi algoritmi (vezi, de exemplu *Kuck 1968*). Limbajele noi proiectate sînt : TRANQUIL, similar ALGOL-ului (Aber et al. 1969), GLYPNIR (Lawrie et al. 1975), ACTUS, similar PASCAL-ului (Perrott 1978) şi CFD FORTRAN (Stevens 1975).

Compania Burroughs a jucat un rol major în dezvoltarea procesoarelor paralele de tip masiv. Începutul l-a marcat Illiac IV, pentru care Burroughs a avut contract în perioada 1969—1973, a continuat cu procesorul paralel PEPE pentru armata americană, (livrarea începînd cu 1976) şi a culminat cu anunţarea în 1977 a calculatorului BSP pentru piaţa ştiinţifică, un concurent pentru proiectele pipeline CRAY-1 şi CYBER-205. Vom descrie în continuare relaţiile dintre PEPE, BSP şi ILLIAC IV.

ILLIAC IV a fost proiectat pentru rezolvarea ecuaţiilor diferenţiale parţiale şi poate fi descris ca un masiv 8×8 de elemente procesoare în virgulă mobilă pe 64 biţi, 4-conex ce execută acelaşi flux de instrucţiuni interpretate de o unitate de comandă. PEPE, Parallel Element Processor Ensemble, a fost proiectat, pe de altă parte, pentru conducerea unui sistem de apărare antibalistic format din instalaţii RADAR şi lansatoare de rachete (*Berg* et al. 1972, *Cornell 1972*, *Vick şi Cornell 1978*). PEPE îşi are originile în activităţile de la Laboratorul Bell, Whippany, privind studiul memoriilor distribuite adresate prin conţinut (*Lee şi Paull 1963*) combinate cu procesarea în virgulă mobilă (*Crane şi Githens 1965*) ce au culminat în 1972 cu realizarea maşinii experimentale PEPE 7C cu 16 elemente procesoare pe 32 biţi (*Crane et al., 1972*). Sistemul PEPE construit de Burroughs este format din 288 elemente procesoare slab interconectate, fie-

care conţinînd 3 procesoare (unul pentru recepţionarea semnalului RADAR, unul pentru calcul şi unul pentru emisia semnalelor de comandă) conduse de 3 unităţi specializate, cîte una pentru fiecare procesor. Cele 3 unităţi de comandă erau conectate la 3 canale de I/E standard ale unui calculator CDC 7600 care activa ca sistem gazdă. Fiecare ţintă identificată de sistem este urmărită de un element procesor şi deoarece nu există legături între ţinte (rachete), nici elementele procesoare nu erau interconectate direct. Comunicaţiile necesare între elementele procesoare se realizau via memoria unităţii centrale. Deoarece sistemul era nestructurat s-a ales termenul de *ansamblu*. Cum fiecare element procesor executa 1 Mflop/s, întregul sistem putea realiza maxim 288 Mflop/s. Estimări mai realiste pentru problemele practice au condus la cifra de 100 Mflop/s (Vick şi Cornell 1978). O versiune cu 36 elemente procesoare a fost livrată Centrului de cercetări avansate BMDATC din Huntsville, Alabama, în 1976.

Unul din inconvenientele legăturilor limitate de transmisie între procesoarele 4-conexe este timpul mare necesar pentru stabilirea unei comunicaţii între cele 64 elemente procesoare şi cele 64 blocuri de memorie în masiv (ILLIAC IV). Burroughs a redus în varianta comercială, BSP (Jensen 1978, Austin 1979), numărul procesoarelor la 16 şi numărul blocurilor de memorie la 17. Numărul mai mic face posibilă stabilirea conexiunilor prin intermediul unei reţele de aliniere („alignment network") între orice procesor şi orice bloc de memorie. Alegerea unui număr prim de blocuri de memorie diferit de cel al procesoarelor, permite utilizarea algoritmilor de mapare care reduc numărul conflictelor de acces la memorie ce intervin în manipularea matricelor. Elementele procesoare sînt organizate serial cu un timp de execuţie a adunării sau înmulţirii de 320 ns pentru obţinerea a 16 rezultate (unul în fiecare element procesor). Suprapunerea atentă a operaţiilor de citire, scriere şi aritmetice împreună cu asigurarea legăturilor complete între elementele procesoare şi blocurile de memorie elimină cele mai multe strangulări; de aici atingerea a 50 Mflop/s pentru rezolvarea celor mai multe probleme. BSP este analizat în detaliu în paragraful §3.4.3., deşi a fost retras de pe piaţă în 1980, înainte de a se vinde cel puţin un exemplar.

ARPA şi NASA au fondat Institutul de calcul avansat (IAC) pentru realizarea lui ILLIAC IV, iar în 1977 acelaşi institut a publicat propunerile pentru realizarea unei maşini denumită PHOENIX (Feierbach şi Stevenson 1976b) care să înlocuiască în anii '80 calculatorul ILLIAC IV. IAC a stabilit necesitatea realizării unei maşini de 10 Gflop/s pentru a rezolva probleme de aerodinamică. Calculatorul PHOENIX poate fi descris ca format din 16 ILLIAC IV, fiecare executînd un flux unic de instrucţiuni. Dacă fiecare element procesor produce un rezultat la 100 ns (o presupunere rezonabilă pentru tehnologia anilor '80) cele 1024 elemente procesoare vor executa 10 op/secundă.

NASA a mai încheiat 2 contracte de cercetare cu CDC şi Burroughs pentru maşini care să înlocuiască calculatorul ILLIAC IV şi care să constituie sisteme de simulare aerodinamică numerică (NASF) pentru mijlocul anilor '80 (Stevens 1979). Proiectul CDC se bazează pe un CYBER 205 cu 4 unităţi pipeline ce lucrează sincronizat, plus a cincea unitate pipeline ca rezervă. Fiecare unitate pipeline poate produce un rezultat pe 64 biţi sau 2 pe 32 biţi la fiecare 8 ns. Oricum, fiecare rezultat poate fi produs prin

3 operații, ceea ce duce la o viteză de 3 Gflop/s. Mai există în sistem și un procesor scalar rapid cu un tact de 16 ns. Proiectul lui Burroughs poate fi privit prin contrast ca o versiune actualizată a arhitecturii BSP, cu 512 elemente procesoare conectate la 521 blocuri de memorie (numărul prim superior cel mai apropiat). Spre deosebire de ILLIAC IV, fiecare element procesor are propria unitate de execuție a instrucțiunilor. Fluxul de instrucțiuni poate fi executat de un element procesor în secvențe diferite ce depind de anumite condiții specifice datelor pe care le prelucrează unitatea respectivă. Cu un timp de execuție a adunării în virgulă mobilă de 240 ns, a înmulțirii în 360 ns și 512 elemente procesoare în sistem se poate atinge 1—2 Gflop/s. S-a propus utilizarea tehnologiei ECL, cu o perioadă a tactului de 40 ns.

Proiectul inițial al calculatorului SOLOMON (Slotnick et al. 1962) prevedea un masiv de 32×32 procesoare pe un bit, fiecare cu o memorie locală de 4096 biți, care executau operații aritmetice pe 1024 numere în paralel, bit cu bit. Această descriere se apropie și de modelul pilot ICL DAP, inițiat în 1972 și lansat în producție în 1976 (Flanders et al. 1977, Reddaway 1979, 1984). Primul exemplar produs a fost instalat la colegiul QUEEN MARY în 1980. Acest sistem cuprindea un masiv de 64×64 elemente procesoare care formau un modul de memorie pentru calculatorul gazdă ICL 2980. Ca și la SOLOMON și ILLIAC IV, elementele procesoare sînt 4-conexe. DAP-ul original folosea integrarea pe scară redusă, cu 16 elemente procesoare pe o placă. În §3.4 se va analiza în detaliu arhitectura sistemului ICL DAP. În octombrie 1986 s-a fondat compania Active Memory Technology (AMT) pentru a dezvolta în continuare conceptul DAP. Ea a produs procesoarele VLSI DAP 500 ca opțiune la stațiile de lucru (§ 3.5).

O caracteristică importantă a proiectului ICL DAP este că logica unui element procesor și memoria sa se află pe aceeași plachetă. O dezvoltare posibilă și atractivă ar fi realizarea unui cip VLSI care să includă aceste circuite. În viziunea lui von Neumann memoria și unitatea centrală sînt atît conceptual cît și fizic situate în unități diferite (chiar sertare diferite), ceea ce poate duce la strangulări severe dacă traficul dintre ele nu este adecvat. La ICL DAP partea de execuție (logica) este distribuită în memorie, adiacent informațiilor de prelucrat. Tehnologia VLSI care permite implementarea a 10 porți/cip determină ca distribuția logicii în memorie să fie o propunere practică. Într-adevăr, aceasta a devenit aproape o necesitate datorită problemelor legate de interconectarea cipurilor. De exemplu, AMT DAP 510 este un masiv 32×32 cu 64 elemente procesoare pe un cip VLSI.

Stone (1970) a propus un calculator cu logica de comandă la nivelul memoriei, adică fiecare segment de 16 KB de memorie cache a sistemului IBM 360/85 ar fi trebuit să aibă funcții logice specializate, de unde execuția operațiilor în paralel pe 16 segmente. Acest concept poate fi folosit atît ca memorie asociativă, cît și pentru execuția de operații aritmetice simple pe 16 elemente în paralel. S-a făcut propunerea, într-un sens limitat, pentru un DAP cu 16 elemente procesoare fără conexiuni între ele. Kautz (1971) a propus un calculator cu logică în memorie mai integrat, denumit Augmented Content Addressed Memory (ACAM) cu circuite specializate

pentru sortare, inversarea matricelor, transformata Fourier rapidă, corelaţii etc. Circuitele LSI urmau să fie masive celulare cu logică universală ce implementau în hard algoritmi pentru calcul paralel : transformata Fourier rapidă (Pease 1968) şi inversarea matricelor (Pease 1967).

1.1.5. Calculatoare ortogonale şi asociative

O altă lucrare de rezonanţă în istoria paralelismului este cea a lui Shooman (1960) denumită „Parallel Computing with Vertical Data", în care descrie organizarea unui calculator „ortogonal" (vezi şi *Shooman 1970*). Unităţile aritmetice ale unui calculator serial convenţional îşi extrag datele din memorie sub forma cuvintelor (de exemplu, numere în virgulă mobilă de 32 biţi), iar prin 1960 cele mai multe calculatoare ştiinţifice procesau biţii unui cuvînt în paralel. O astfel de procedură poate fi denumită serial pe cuvînt şi paralel pe biţi. Shooman a recunoscut că multe probleme ce implică regăsirea informaţiilor necesită testarea numai cîtorva biţi ai cuvîntului şi astfel prelucrarea serială pe cuvînt nu este eficientă. Shooman a propus ca memoria să fie accesată şi în direcţie ortogonală. adică de la fiecare cuvînt bitul de acelaşi rang (bit slice). Dacă biţii memoraţi sînt interpretaţi ca un masiv bidimensional, cu biţii cuvîntului k formînd linia orizontală k, atunci biţii de rang l ai tuturor cuvintelor (bit slice l), formează coloana l. În calculatoarele ortogonale fiecare element procesor corespunde la un cuvînt de memorie şi, astfel, biţii ce formează un bit slice pot fi prelucraţi în paralel. Acest procedeu este denumit procesare serială pe bit şi paralelă pe cuvînt. Calculatorul ortogonal asigură o „unitate orizontală" pentru realizarea operaţiilor serial pe cuvînt, paralel pe bit şi o „unitate verticală" separată pentru operaţii serial pe bit, paralel pe cuvînt.

Ideea testării în paralel a tuturor cuvintelor a condus la realizarea memoriilor *asociative* sau *adresabile prin conţinut*, la care o informaţie-cuvînt este identificată în memorie printr-o secvenţă de biţi din componenţa sa care satisface un criteriu de căutare, mai degrabă decît prin adresa fizică. La o memorie asociativă pură, informaţiile nu pot fi accesate prin adresa fizică din memorie. Totuşi, multe sisteme asigură ambele forme de adresare. Multitudinea sistemelor ce se bazează pe memorii asociative au fost analizate de Thurber şi Wald (1975) şi Yau şi Fung (1977), iar cititorul mai poate consulta monografiile lui Foster (1976) şi Thurber (1976). Noi vom descrie aici numai 2 exemple, OMEN şi STARAN care sînt deja comercializate.

Seria OMEN (Orthogonal Mini EmbedmeNt) a reprezentat implementarea comercială a conceptului de calculator ortogonal, cu aplicaţii la analiza semnalelor. Aceste calculatoare sînt descrise de Higbie (1972). Seria OMEN-60 folosea un PDP-11 ca unitate aritmetică orizontală şi un masiv de 64 elemente procesoare pentru unitatea verticală asociativă, care lucra mai degrabă byte slice, decît bit slice. Opţional se asigură fie o unitate aritmetică serială pe bit şi 8 biţi de memorie pentru fiecare element procesor, fie o unitate hardware în virgula mobilă cu 8 registre de 16 biţi şi 5 registre de mascare. Logica implementată permitea inversarea

ordinii octeților într-un slice sau, realizarea conexiunii „perfect shuffle" sau „barrel shift".

Un alt produs al conceptului de calculator ortogonal este **STARAN** (*Batcher 1979*) produs de Goodyear, proiectat între anii 1962 și 1972 și din care s-au vîndut pînă în 1976 4 bucăți. Modelul STARAN standard este format din 4 masive a cîte 256 elemente procesoare pe un bit și memorie de la 64 KB la 64 MB. Conducerea sistemului era asigurată de un PDP 11. Spre deosebire de SOLOMON, STARAN nu are memoria distribuită procesoarelor; între elementele procesoare și memoria centrală este realizată o rețea programabilă, denumită FLIP. Un slice de 256 biți este extras din memorie sub controlul programului într-o modalitate de căutare după pattern. Patternul selectat poate trata, de exemplu, memoria ca un masiv multidimensional cu un număr variabil de dimensiuni sau extrage datele în modul cerut de transformata Fourier rapidă sau alți algoritmi numerici. Conexiunea între elementele procesoare se realizează prin comutarea slice-urilor de 256 biți la nivelul rețelei. FLIP. ETAFAN, ca și alte calculatoare orientate pe bit, are performanțe mai bune pentru operații logice sau aritmetice cu numere mici. O aplicație particulară este prelucrarea numerică a imaginilor, caz în care imaginea este împărțită în milioane de pixeli reprezentați pe 6—12 biți. Primul STARAN a fost livrat bazei militare ROME AIR FORCE tocmai pentru aplicații de prelucrare a imaginilor și controlul traficului aerian.

STARAN și DAP au fost reunite în propunerea de către firma Goodyear Aerospace Corp. a unui procesor paralel masiv (MPP) format dintr-un masiv de 132×128 elemente procesoare 4-conexe. Mașina este proiectată special pentru prelucrarea imaginilor din satelit, cu o viteză de 1000000 pixeli/s. Operațiile cu numere întregi reprezentate pe 8—12 biți se realizează cu o viteză de cîțiva Gflop/s, iar operațiile în virgulă mobilă pe 32 biți, la 200—400 Mflop/s.

1.1.6. Masive de procesoare și procesoare-extensii

O altă direcție de evoluție a calculatoarelor ce implementează paralelism este cea a calculatoarelor specializate, relativ ieftine, proiectate pentru prelucrarea masivelor de date, de unde și denumirea de procesoare masiv. Observăm că, în acest context, denumirea lor *nu* înseamnă că aceste calculatoare sînt masive bidimensionale de procesoare, dimpotrivă, în cele mai multe proiecte se implementează conceptul pipeline. Astfel de calculatoare necesită, dar nu întotdeauna, un calculator gazdă, de unde denumirea de *procesoare-extensii*. Domeniile de utilizare sînt analiza semnalelor, a datelor seismice, iar algoritmul cel mai des implementat este al transformatei Fourier rapide. Primele sisteme de acest tip au fost minicalculatoarele pe 16 biți care implementau accesul paralel la instrucțiuni și date. Tehnologia bipolară a permis execuția a 5×10^4 op/s. Următoarea fază a fost marcată de realizarea „unităților funcționale" pentru FFT și algoritmi similari. La aceste unități paralelismul a fost introdus pe scară largă, ca de exemplu accesul în paralel la mai multe module de memorie pentru programe, date și coeficienți,utilizarea mai multor unități aritmetice inter-

conectate în modul cerut de rezolvarea problemei. Unitățile funcționale erau microprogramate, dar, deși inflexibile, realizau o creștere a performanțelor de 10 ori, atingînd 0,5 Mflop/s.

În 1973 a apărut o nouă generație de sisteme masiv cu mai multe unități procesoare interconectate cu un număr redus de magistrale și care lucrează asincron sub comanda unei unități centrale. Se ating 5 Mop/s, deși modul asincron creează dificultăți de sincronizare și nereproducerea condițiilor la verificarea programelor.

De departe succesul cel mai mare l-a avut sistemul AP-120B al firmei FLOATING POINT SYSTEMS care a însemnat o întoarcere la principiul funcționării sincrone (Harte 1979). Această companie a fost fondată în 1970, a lansat în producție sistemul menționat în 1976, iar în 1985 erau deja instalate aproximativ 5000 sisteme. AP-120B, care poate fi conectat atît la minicalculatoare cît și la calculatoarele mari ca gazdă, execută operațiile aritmetice în virgulă mobilă pe 38 biți în pipeline separate pentru adunare și înmulțire, iar calculul adreselor și evidența buclelor se realizează într-o unitate separată pe 16 biți. 3 memorii (pentru date, tabele și programe) și 2 blocuri (scratch pad) de registre rapide cu acces multiplu se află între fiecare memorie și fiecare unitate aritmetică. Instrucțiunile mașinii sînt pe 64 biți. Astfel, s-a realizat un procesor pentru masive de date relativ ieftin. În mod obișnuit s-au atins 5—10 Mflop/s, la un cost de aproximativ 50000$ (preț 1980). Acest sistem este deosebit de avantajos pentru industrie, atît ca performanțe cît și ca preț. Din multe puncte de vedere AP-120B poate fi considerat ca sistemul CRAY-1 al „omului sărac", fiind de 10 ori mai lent dar și de 50 ori mai ieftin. Arhitectura de ansamblu a celor 2 mașini este similară, amîndouă folosind unități funcționale pipeline independente. În 1980 FPS a anunțat FPS-164 FPS-164 Attached Processor, o versiune pe 64 biți și memorie operativă extinsă la 1,5 Mcuv. Viteza maximă de calcul este de aproximativ 11Mflop/s (perioada ceasului de 182 ns), dar memoria operativă mai mare elimină necesitatea unui calculator gazdă, de unde și dispariția timpilor de așteptare datorită transferurilor lente. În mod obișnuit viteza de calcul variază între 1 și 8 Mflop/s, funcție de problema și maniera de programare. Deoarece nu există instrucțiuni vectoriale, se folosesc bucle scalare. În 1985 s-a anunțat o implementare a lui FPS 164 în tehnologie ECL. Această mașină, denumită FPS 264, este de la 4 la 5 ori mai rapidă decît FPS 164, dar identică în rest.

În anul 1984 s-a anunțat o nouă înbunătățire a calculatorului FPS 164, FPS 164/MAX, ceea ce înseamnă Matrix Algebra Accelerator. Această mașină folosește un FPS 164 ca master, la care se pot adăuga pînă la 15 plăci MAX, fiecare echivalentă cu 2 CPU FPS 164. Se obține un echivalent de 31 CPU FPS 164 ce pot atinge o viteză maximă de 341 Mflop/s. În §2.5.7. și §2.5.8 se tratează despre FPS 164, respectiv FPS 164/MAX.

1.1.7. Propunerea japoneză pentru generația a cincea

În octombrie 1981 japonezii au șocat comunitatea specialiștilor în calculatoare cu perspectiva lor asupra calculatoarelor anilor 1990 pe care le-au denumit calculatoare de generația a cincea (Feigenbaum și McCorduck

1983). Ideile lor au fost anunțate lumii la o conferință internațională desfășurată la Tokyo, și denumită „Sisteme de calcul de generația a cincea" (FGCS) (Motooka 1982, 1984). Japonezii consideră ca principale aplicații pentru calculatoare în anii 1990 sistemele expert și sistemele bazate pe cunoștințe (KBS). Aceste sisteme folosesc o bază cu reguli ce reprezintă cunoștințele unui expert, pentru extragerea unor concluzii și inferențe pe baza unor fapte furnizate interactiv de către utilizator. Comportarea sistemului este asemănătoare unei consultații între un expert uman și cineva care are o problemă ce solicită cunoștințe de expert. Pînă acum aplicațiile cu succesul cel mai mare s-au înregistrat în domenii limitate, cu cunoștințe bine definite, ca de exemplu, diagnoza medicală în domenii specifice. Alte exemple se referă la traducerea de texte, întreținerea mașinilor și analiza economică. Unele firme sînt îngrijorate de pierderea experienței unui angajat care se pensionează. O soluție parțială ar fi înregistrarea cunoștințelor sale într-un sistem expert.

Pentru a se atinge obiectivele de mai sus s-a stabilit că un FCGS ar trebui să fie format din trei părți : o mașină bază de cunoștințe cu o memorie de 100 GB ; o mașină inferențială și de rezolvare a problemelor care să gestioneze baza de cunoștințe și să răspundă întrebărilor puse ; și o interfață inteligentă pentru comunicarea cu utilizatorul prin sunet și imagine. Ministerul japonez al industriilor și comerțului (MITI) colaborează cu 8 firme electronice (inclusiv Fujitsu, Hitachi, NEC, Mitsubishi, OKI și Toshiba) în cadrul unui proiect național pe 10 ani, pentru a construi pînă în 1992 un FGCS. Proiectul este condus de Prof. Motoroka (*) de la *Universitatea din Tokyo și are un buget de aproximativ 400 mil. $.*

Mașina inferențială este apreciată după numărul de inferențe logice executate pe secundă (LIPS) și se estimează că un calculator de generația a cincea trebuie să execute între 100 MLIPS și 1 GLIPS. Calculatoarele actuale pot realiza între 0,01 și 0,1 MLIPS, deci este necesară o creștere a vitezei de execuție de 4 ordine de mărime. O astfel de creștere se poate obține, probabil, numai printr-o combinație între o tehnologie, arhitectură și metode avansate. S-a dedus că numai creșterea vitezei dispozitivelor electronice nu va fi suficientă și că trebuie ca sistemele de calcul să devină mai paralele, să folosească arhitecturi ca cea denumită *data-flow*, care exploatează în mod automat paralelismul inerent al problemei. În astfel de calculatoare, o operație are loc imediat ce operanzii ei sînt disponibili și există o unitate funcțională liberă (vezi §3.2.2. și Treleaven 1979, Treleaven, Brownbridge și Hopkins 1982 și Dennis 1980). Spre deosebire de calculatoarele controlate prin flux de instrucțiuni, denumite calculatoare *control-flow*, aici singurele restricții în ordinea de execuție sînt cele impuse de dependențele de date. Cu aceste limite, ce pot fi identificate de către hardware, un calculator *data-flow* poate executa în paralel atîtea operații cîte unități funcționale are. De o importanță mult mai mare este faptul că programatorul nu trebuie să identifice paralelismul problemei și deci nici să-l programeze explicit în fluxul de instrucțiuni. Utilizarea unor nivele înalte de paralelism prin multiplicare este necesară pentru exploatarea avantajelor oferite de producția de masă VLSI.

---

* Între timp decedat (Nota red.)

Un alt aspect important al conceptului generaţiei a cincea este îmbunătăţirea considerabilă a interfeţei între utilizatorul uman şi calculator, cunoscută cu denumirea de interfaţa om-maşină (MMI). Calculatoarele curente sînt ineficiente în recunoaşterea vorbirii, manipularea textelor, comunicarea grafică şi prelucrarea imaginilor; toate acestea sînt caracteristici ale unei interfeţe bune pentru utilizator. Se acordă o mare importanţă progreselor în aceste domenii, care conduc tot la calculul paralel; întradevăr, deja calculatoarele înalt paralele, SIMD, ca ICL DAP şi STARAN s-au dovedit foarte eficiente în prelucrarea imaginilor. În afara comunicării prin imagini şi sunet, MMI ar trebui să utilizeze maşina inferenţială, pentru a apare utilizatorului cît mai „inteligentă".

Proiectul generaţiei a cincea de calculatoare nu are ca obiectiv creşterea performanţelor de calcul numeric, deoarece acesta este subiectul unui proiect separat. MITI a lansat în 1982 un alt proiect naţional pentru dezvoltarea „Sistemelor de calcul de mare viteză pentru scopuri ştiinţifice şi tehnologice", urmînd ca în 1989 să se producă un supercalculator de 10 Gflop/s (Kashiwagi 1984). Proiectul este condus de dr. H. Kashigawi de la Laboratorul de electrotehnică, şi are un buget de 100 mil. $. La ele colaborează cu cele 6 firme electronice deja menţionate în legătură cu proiectul generaţiei a cincea. Pentru jusificarea programului s-au menţionat aplicaţii de prognoză a vremii, cercetările în aerodinamică şi fizică nucleară. Proiectul constă în cercetări în domeniul arhitecturii calculatoarelor paralele şi soft-ului al unor noi circuite logice cu viteză şi integrare mare şi, în final, în construirea şi evaluarea supercalculatorului. Obiectivele sînt noi dispozitive logice cu timp de propagare/poartă de 10 ns (prin folosirea joncţiunii Josephson lucrează la temperatura heliului lichid, tranzistorul HEMT, ce lucrează în azot lichid), porţi cu timp de propagare de 3ns. (pe bază de arseniură de galiu), ca şi noi circuite de memorie cu timp de acces de 10 ns. Nivelul de integrare urmărit pentru porţile logice este de 3000 porţi/cip, iar pentru memorie de 16 Kb/cip. În plus faţă de proiectul naţional, în cursul anilor 1984/5 au fost lansate supercalculatoare produse de diverse companii. Acestea sînt proiecte cu mai multe unităţi pipeline, similare cu CRAY-1 şi CYBER 205. *Fujitsu FACOM VP-100/VP-200, HITACHI S-810* model 10/model 20 şi *NEC SX-1/SX-2* au performanţele maxime de 533 Mflop/s, 630 Mflop/s şi, respectiv, 1300 Mflop/s. În cadrul unor universităţi şi institute de cercetare naţională s-au construit mai multe maşini experimentale, cu arhitecturi diferite (masive de procesoare, calculatoare MIMD şi data-flow), de aceea va fi interesant de văzut care arhitectură va fi aleasă în final pentru supercalculatorul naţional.

1.1.8. Calculatoare MIMD

Se spune uneori că, datorită progreselor tehnologiei VLSI şi a microprocesoarelor ieftine, anii '80 vor fi decada calculatoarelor MIMD. Astfel de calculatoare, cu fluxuri multiple de instrucţiuni/fluxuri multiple de date, sînt controlate de mai multe fluxuri de instrucţiuni. Noi limităm aici termenul la sistemele strîns conectate, în cadrul cărora fluxurile de instrucţiuni pot fi programate să coopereze împreună pentru a rezolva o

singură problemă. Excludem, în acest mod, calculatoarele slab conectate care comunică prin mesaje transmise prin intermediul unor rețele (de exemplu, ARPANET), sau configurațiile multiprocesor care sînt ascunse utilizatorului, ca cele 4 procesoare ale calculatorului IBM 3081. Deja am întîlnit exemple de sisteme MIMD pe scară mică sub forma calculatoarelor CRAY X-MP cu 4 CPU și CRAY-2, sau ETA[10] cu 8 CPU. Totuși, de obicei calculatoarele MIMD sînt gîndite ca ansambluri de mai multe calculatoare (de obicei mini sau micro) și module de memorie, conectate cu un switch sau rețea. Din punct de vedere logic, același efect se poate obține prin transmiterea fluxurilor multiple de instrucțiuni unei singure unități de procesare, așa cum se procedează la Denelcor HEP cu o singură unitate PEM. Ambele sînt exemple de calculatoare MIMD, deoarece au fluxuri multiple de instrucțiuni. În momentul de față sînt atît de multe arhitecturi MIMD propuse, încît includerea aici a unei liste complete, chiar dacă ar fi posibilă, ar fi inadecvată și plictisitoare. Deoarece s-au construit și operează cu succes numai puține calculatoare, le-am ales numai pe acestea pentru a le descrie aici. Un studiu mai amănunțit este prezentat în lucrările Hockeney (1985b, d, 1986).

(i) *Calculatoarele C. mmp și Cm* de la Carnegie-Mellon*

Unul din proiectele inițiale de calculatoare MIMD dintre cele mai ambițioase a fost calculatorul C. mmp de la Universitatea Carnegie-Mellon (Wulf și Bell 1972, Mashburn 1982). Acesta este format din 16 minicalculatoare DEC PDP-11 conectate la 16 module de memorie printr-o rețea cu accesuri încrucișate. Rețeaua asigură posibilitatea unei legături electronice directe între orice calculator și orice modul de memorie (vezi fig. 3.8). Calculatoarele comunică prin partajarea unui spațiu de adrese logice de 32 MB, din care 27 MB au fost implementate de modulele de memorie. De asemenea, toate calculatoarele sînt conectate la o magistrală comună. Proiectul a fost lansat în 1971, iar mașina a fost terminată în 1975. A rămas operațională pînă în 1980. Succesorul lui, Cm* a fost un concept complet diferit, ce a folosit microprocesoarele disponibile pe piață (*Swan Fuller și Siewiorek 1977, Fuller et al 1978*). Memoria sistemului Cm* este, spre deosebire de situația de la C. mmp, împărțită între microprocesoare fiind, deci locală; o rețea ierarhică asigură comunicarea între microprocesoare. Unitatea de bază este „modelul calculator" care constă dintr-un microprocesor DEC SI-11 cu 64 KB de memorie MOS dinamică și probabil periferice, conectate la magistrala LSI-11. Acest modul poate lucra ca un calculator independent; pînă la 14 module pot fi conectate la o magistrală comună „intra-ciorchine" (intra-cluster), pentru a forma un ciorchine strîns conectat cu transferul datelor prin acces direct la memorie. Sistemul este construit prin conectarea împreună a ciorchinelor prin 2 magistrale inter-ciorchine (intercluster) pentru a forma o rețea slab conectată cu transferul datelor prin comutarea pachetelor. Ambele tipuri de magistrale sînt controlate de calculatoare microprogramate. Natura conectării între modulele calculator este invers proporțională cu timpul de transfer al datelor între ele, și aceasta poate varia cu un factor de 10, funcție de pozițiile relative ale modulelor. Transferurile cu memoria locală a aceluiași modul durează 3 microsec., cele între module diferite ale aceluiași ciorchine durează 9 microsec., iar cele între ciorchini diferiți durează 26

microsec. Toate calculatoarele componente ale Cm* partajează un spaţiu de adresare virtual comun de 256 MB, iar sistemul este expandabil prin adăugarea unor noi ciorchini. Se speră să se exploateze avantajele oferite de o viitoare implementare VLSI a arhitecturii, cu mai multe sute sau chiar mii de calculatoare. Proiectarea detaliată hardware a început în 1975 şi, un ciorchine cu 10 calculatoare a devenit operaţional în 1977. Acum este operaţional un sistem Cm* cu 5 ciorchini, cu un total de 50 calculatoare, propus ca instrument pentru testarea altor proiecte de calculatoare MIMD. Atît C.mmp. cît şi Cm* sînt descrise de Satyanaranyanan (1980), iar experienţa de operare poate fi găsită în lucrarea lui *Jones şi Schwarz* (*1980*).

(ii) *Experimentele britanice*

Sistemele MIMD experimentale dezvoltate la sfîrşitul anilor '70 şi începutul anilor '80 sînt prea numeroase pentru a le enumera pe toate. Totuşi, următoarele sisteme sînt tipice. La Universitatea Loughborough s-au construit sub conducerea Prof. Evans sisteme MIMD cu memorie comună, care au fost apoi folosite pentru proiectarea unor algoritmi MIMD paraleli (Barlow, Evans şi Shanehchi 1982). Primul sistem folosea două minicalculatoare Interdata 70, partajînd 32 KB ai spaţiului lor de adresare. Mai tîrziu, în 1982, au fost folosite 4 microcalculatoare TI 990/10 cu memorie comună pentru a constitui un procesor paralel asincron (Barlow et al 1981). Un sistem MIMD mai mare denumit CYBA-M, a fost construit la Universitatea Manchester, în cadrul Institutului pentru ştiinţă şi tehnologie (UMIST), sub conducerea Prof. Aspinall. El constă din 16 microprocesoare 8080 ce folosesc o memorie multiport (Aspinall 1977, 1984, Dagless 1977). Alte proiecte interesante sînt calculatoarele DataFow de la Manchester, maşina reducţionistă ALICE de la Imperial College şi proiectele RPA şi Supernode de la Universitatea Southampton. Ultimele două sînt prezentate în capitolul 3.

(iii) *Ultracalculatorul de la Universitatea New York şi IBM RP3*

Cele mai multe proiecte britanice nu au avut o publicitate bună, deşi au fost construite şi au operat cu succes mulţi ani. Un proiect academic mult mai ambiţios, Ultracalculatorul NYU, a avut o publicitate foarte mare, deşi a rămas mulţi ani numai un proiect de hîrtie. Oricum, lucrările publicate au influenţat teoria şi proiectarea sistemelor MIMD mari. Ca răspuns la potenţialul oferit de tehnologia VLSI, conceptul original al lui Schwartz (1980) se referea la o familie de arhitecturi de calculatoare, denumite ultracalculatoare, care conectau împreună mii de elemente procesoare într-o reţea cu o topologie denumită a amestecului perfect (perfect-shuffle nearest-neighbour — PSNN) — vezi fig. 3.21. Această reţea impune conectarea fiecărui element procesor la cel mult alţi patru şi, s-a arătat, este foarte indicată pentru implementarea multor algoritmi paraleli ce folosesc strategia „divide et impera". Un ultracalculator cu 16 K elemente procesoare poate fi folosit ca un masiv bi-dimensional 128×128, sau ca un masiv tri-dimensional $32 \times 32 \times 16$, sau ca un hipermasiv în 4 dimensiuni $16 \times 16 \times 8 \times 8$. Memoria trebuie distribuită prin reţea cu 132 Kcuv. de 16 biţi de memorie locală pentru fiecare procesor. Circuite separate pentru transferul datelor implementează conexiunile PSNN între PE grupate cîte 8. Într-un proiect ulterior al ultracalculatorului (Gottlieb

et al 1983), memoria locală a fost eliminată, iar memoria operativă s-a concentrat în module de memorie (MM). Elementele procesoare sînt conectate la MM prin comutatoare multi-nivel, sub forma unei rețele Omega (Lawrie 1975). Dacă sînt N procesoare și N module de memorie, sînt necesare log2N nivele. S-a luat în considerare implementarea unei mașini cu 4096 procesoare, pentru anii '90. La început se va construi un prototip cu 64 elemente procesoare pe baza microprocesoarelor și cipurilor de memorie comercializate, și a unui circuit de comutare proiectat chiar de constructorii mașinii.

IBM construiește o versiune de ultracalculator la *Yorktown Heights Research Laboratory*, cu 512 elemente procesoare, denumită Research Parallel Processor Project (RP3). Acesta constă din 4 secțiuni a 64 procesoare fiecare. Prima secțiune este programată să fie operațională în cursul anului 1988.

O trăsătură specifică a ultracalculatorului este posibilitatea execuției unei instrucțiuni ,,extrage și adună'' (fetch-and-add-FA), parțial implementată de nodurile rețelei .Dacă mai multe procesoare doresc să adune datele la aceeași locație de memorie, pot executa în mod independent instrucțiunea FA, probabil chiar în același moment. Hard-ul rețelei combină aceste cereri în momentul cînd apar în noduri și, în final, memoria este incrementată ca și cum toate cererile au fost lansate într-o ordine arbitrară. Astfel, se menține secvența necesară de operații de adunare chiar dacă ordinea, care nu este importantă, nu este cunoscută. Operațiile de extragere și adunare sînt de importanță fundamentală pentru mulți algoritmi MIMD : de exemplu, pentru decrementarea unei variabile de sincronizare a mai multor procese, sau la acumularea sarcinilor într-o simulare de plasmă. Asigurarea acestei facilități în cadrul ultracalculatorului este o inovație importantă, care va crește viteza de execuție a multor programe. Dacă fiecare procesor execută o instrucțiune FA cu aceeași variabilă, toate rezultatele se vor obține în intervalul de timp necesar execuției unei singure instrucțiuni.

(iv) *Proiectul Cedar de la Universitatea Illinois*

Proiectul Cedar, în curs de realizare la Universitatea Illinois de către D. Kuck și echipa sa se referă la realizarea unei mașini de mai mulți gigaflop/s, în anii '90 (Gajski et al 1983). Arhitectura generală este similară ultracalculatorului, dar Cedar nu posedă instrucțiuni extrage-și-adună. Printr-o rețea generală Omega, sînt conectați 16 ciorchini a cîte 8 procesoare, la 256 module de memorie de 4 pînă la 16 Mcuv. fiecare. Intervalul de timp între cererea de acces la memoria generală și disponibilitatea datei la procesor este de aproximativ 2 microsec. Fiecare ciorchine are 8 procesoare cu 16 Kcuv. de memorie locală fiecare. Aceste procesoare sînt pipeline și sînt conectate printr-o rețea locală. Toate locațiile de memorie au un indicator ocupat-liber (ca la Denelcor HEP) pentru a permite sincronizarea între procesoare. S-a planificat ca prototipul Cedar 32 să aibă 4 ciorchini a 8 procesoare și să lucreze cu o perioadă a ceasului de 400 ns. Astfel, fiecare procesor atinge 2,5 Mflop/s cu o tehnologie lentă, rezultînd un total de 80 Mflop/s (comparabil cu CRAY-1) pentru un prototip de birou. Acesta se va putea mări la Cedar 128 cu 16 ciorchini (320 Mflop/s în 1988) și la Cedar 512 cu 64 ciorchini (1,2 Gflop/s-în 1990). O altă

versiune tehnologică pentru o perioadă a ceasului de 40 ns şi 25 Mflop/s pentru un procesor este planificată, cu denumirea Cedar 32H pentru 4 ciorchini (800 Mflop/s în 1989)şi Cedar 128H cu 16 ciorchini (3,2 Gflop/s în 1991). Proiectul a fost lansat în 1983, iar pentru versiunile mai ambiţioase se doreşte colaborarea firmelor industriale. De fapt, planurile expuse anterior nu au fost urmate exact, deoarece în 1985 s-a decis utilizarea calculatorului comercializat Alliant FX/8, ca ciorchine pentru Cedar. Această maşină este descrisă în §1.1.9.

(v) *Proiectul EGPA de la Universitatea Erlangen*
Probabil arhitectura MIMD cea mai originală şi imaginativă a fost dezvoltată sub conducerea Prof. Händler la Universitatea Erlangen, R.F.G., denumită Erlangen General Purpose Array, sau EGPA, (*Handler, Hofmann şi Schneider 1975, Händler 1984*). Conexiunile între calculatoarele sistemului EGPA sînt topologic similare unei piramide. Calculatorul de control C din vîrful piramidei conduce 4 calculatoare B aflate la colţurile bazei sale. Cele 4 calculatoare B au, de asemenea, legături directe între ele după muchiile bazei. Acest sistem cu 5 calculatoare lucrează din 1981, folosind minicalculatoare AEG 80—60, fiecare cu 512 KB de memorie. Sistemul poate fi extins la nivele ulterioare, făcînd fiecare calculator B vîrful unei noi piramide cu 4 calculatoare A la baza sa. S-ar obţine un total de 16 calculatoare, care sînt conectate între ele sub forma unui masiv 4×4 ca la ICL DAP. Avantajul topologiei piramidei EGPA este că o comunicaţie pe trasee scurte poate avea loc foarte eficient, în plan, în timp ce cele pe trasee lungi se realizează tot eficient, prin trimiterea datelor la nivele superioare în piramidă. De exemplu, dacă calculatoarele de la bază formează un masiv $n \times n$, atunci calculatoarele cele mai îndepărtate pot comunica în $2\log_2 n$ paşi prin vîrful întregii piramide, în timp ce în plan ar fi necesari 2n paşi. Sistemul cu 5 calculatoare a fost folosit pentru prelucrarea conturului, în cadrul prelucrării imaginilor, şi s-a dovedit de aproximativ 3 ori mai rapid decît un singur calculator de acelaşi tip (Herzog 1984). Dacă se continuă dezvoltarea ierarhică cu un nou nivel, fiecare calculator A devenind vîrful unei noi piramide, la cele 4 nivele vor fi 1, 4, 16, şi respectiv, 64 calculatoare, ceea ce înseamnă un total de 85 calculatoare. Aceasta este baza proiectului Erlangen Multiprocessor 85 (Händler et al 1985).

(vi) *Livermore S-1*
De departe, proiectul MIMD cel mai ambiţios este calculatorul Livermore S-1 (Widdoes şi Correll 1979, Farmwald 1984). Finanţat de U.S. Navy şi Departamentul energiei, sprijinit de E. Teller, proiectul complet S-1 cuprinde 16 calculatoare vectoriale pipeline din clasa CRAY-1, conectate la 16 blocuri de memorie printr-o reţea cu accesuri încrucişate, care asigură o legătură logică directă între fiecare calculator şi fiecare bloc de memorie. De aceea, S-1 poate fi privit ca o versiune „matură" a calculatorului C. mmp. Se estimează o performanţă generală echivalentă cu cea a 10 CRAY-1, adică aproximativ 1 Gflop/s. Fiecare din calculatoarele componente (denumite uniprocesoare) are o memorie cache pentru date (64 KB) şi una pentru instrucţiuni (16KB) pentru a limita traficul prin reţea. De asemenea, toate calculatoarele sînt conectate direct la o magis-

trală comună (cutia de sincronizare) pentru a facilita transferul unui număr redus de date, ca și sincronizarea rapidă a mesajelor dintre ele. Fiecare modul de memorie poate avea capacitatea de maximum 1 GB (2^{30}), ceea ce produce un total de 16 GB (baiți cu 9 biți). Oricum, programatorul lucrează cu o memorie virtuală uniformă de pînă la 2 GB. Mecanismul de paginare hard translatează adresele virtuale în adrese fizice, fără necesitatea intervenției programatorului. Operațiile aritmetice obișnuite se fac în virgulă mobilă pe 36 biți, dar se pot executa și cu numere de 18 sau 72 biți.

Pentru anumite funcții matematice (sin, exp, log) există instrucțiuni care consumă timpul necesar pentru 2 înmulțiri. În plus fața de instrucțiunile vectoriale element-cu-element, se pot executa cu instrucțiuni dedicate : transpunerea matricelor, înmulțirea matricelor și transformarea Fourier rapidă. Nu există registre vectoriale, și toate operațiile cu vectori folosesc memoria operativă, considerată ca un spațiu continuu de elemente succesive, ca la CYBER 205. De asemenea, nu există instrucțiuni de dispersare/grupare pentru asamblarea vectorilor continui din date neordonate, deși se poate folosi instrucțiunea pentru transpunerea matricelor, dacă datele sînt ordonate. Proiectul S-1 facilitează aplicarea continuă a noilor tehnologii în cadrul unei aceleiași arhitecturi. Un uniprocesor MARK I, ce este implentat în tehnologie ECL 10K MSI, a fost lansat în execuție în 1977, urmat de MARK II A cu tehnologie ECL 100K în 1983. MARK V înseamnă un supercalculator pe o capsulă.

(vii) *MIDAS*

Sistemul Modular Interactive Data Analysis System (MIDAS) de la Universitatea Berkeley, California, (Maples et al 1981) este un sistem ierarhic constituit dintr-un calculator primar care conduce mai multe calculatoare secundare, fiecare controlînd la rîndul lui un masiv multiprocesor (MPA). Calculatorul primar realizează controlul sistemului, comunică cu utilizatorii și alocă resursele fiecărui job.Calculatoarele secundare rezolvă probleme individuale, execută părțile secvențiale și alocă părțile paralele pentru execuție MPA. La nivelul cel mai de jos, MPA sînt ciorchini de cîte 8 elemente procesoare (Modcomp 7860), care formează nucleul de calcul al sistemului. Fiecare MPA are un procesor pentru intrare și unul pentru ieșire, ca și o rețea în cruce pentru conectarea la 16 module de memorie, a cîte 256 KB fiecare. Alocarea unui modul de memorie la un procesor poate fi schimbată în 50 ns, după care el poate fi adresat direct de către procesor. Deci, MPA este un calculator MIMD cu memoria partajată printr-o rețea cu accesuri încrucișate. Un prototip cu 4 procesoare și 8 module de memorie este operațional din ianuarie 1982, iar un subsistem complet cu un calculator primar, un calculator secundar și un MPA este operațional din februarie 1983. Modcomp 7860 atinge aproximativ 85% din viteza de calcul a lui VAX 11/780, care echivalează cu aproximativ 0,3 Mflop/s, de unde o performanță generală de aproximativ 3 Mflop/s (echivalentă aproximativ cu cea a unui CDC 7600), pentru un sistem cu un MPA. Acest sistem a fos folosit pentru a rezolva o serie de probleme de fizică numerică și nucleară.

(viii) *Finite Element Machine de la NASA*

Finite Element Machine (FEM) proiectată în cadrul NASA (Jordan 1978) este un masiv MIMD bidimensional controlat de un minicalculator TI 990. Elementele procesoare sînt microcalculatoare TI 9900 care au coprocesoare aritmetice în virgulă mobilă, AM 9512, 32 KB RAM şi 16 KB ROM. Căi de date de 1 bit conectează fiecare procesor cu cei mai apropiaţi 8 vecini (atît în direcţie ortogonală, cît şi diagonală). De asemenea, toate procesoarele sînt conectate la o magistrală comună pe 16 biţi, care poate sesiza starea a 8 registre de condiţii din fiecare procesor. Pe baza acestor indicatori se pot realiza diverse funcţii logice ca „toate", „oricare", folosite pentru sincronizarea operaţiilor masivului. Proiectul maşinii descrie un masiv de 6×6 procesoare. Iniţial, s-a construit un masiv 4×2, operaţional în 1983. La acesta s-a adăugat în 1984 un al doilea masiv 4×2, formîndu-se o structură 4×4. Masivul iniţial a fost folosit intensiv, cu rezultatul unei experienţe de operare cu sistemele MIMD.

(ix) *Hipercuburile* (*Cosmic Cube, Intel iPSC, FPS T-Series*)

Hipercuburile binare au fost studiate pe larg ca reţele de interconectare posibile pentru calculatoarele MIMD, odată cu publicarea importantei lucrări a lui Pease (1977), denumită „The indirect binary n-cube microprocesor array". Totuşi, aceste maşini nu au fost realizate practic pînă la construirea calculatorului Cosmic Cube la Caltech, de către G. Fox şi şi C. Seitz în 1984. Ca rezultat al publicităţii făcute acestei maşini, au apărut mai multe versiuni comerciale şi se observă o dezvoltare rapidă a acestei direcţii.

Punctul, linia, pătratul şi cubul cu noduri la colţuri şi conexiuni de-a lungul muchiilor reprezintă reţelele hipercub de ordinul 0, 1, 2 şi, respectiv, 3. Fiecare nou cub este construit prin considerarea a 2 copii ale hipercubului de ordin imediat inferior, care sînt unite la colţuri (noduri). De exemplu, hipercubul de ordinul 4 este format din două cuburi prin conectarea colţurilor corespunzătoare. În această secvenţă, hipercubul de ordin d are $n=2^d$ noduri şi $d=\log_2 n$ conexiuni la fiecare nod. Astfel, unul din aspectele atrăgătoare ale acestei scheme de conectare este că numărul conexiunilor creşte relativ lent odată cu dimensiunea hipercubului. Un alt aspect atrăgător este că hipercubul de ordin d are, evident, ca subseturi conexiunile tuturor hipercuburilor de ordin inferior, ca şi toate conexiunile necesare pentru execuţia unei transformate Fourier rapidă cu n date. Adică, dacă corespunzător celor n date se atribuie o valoare fiecărui nod al unui hipercub de ordinul d, atunci datele combinate în orice etapă a execuţiei algoritmului FFT sînt permanent în noduri adiacente. Vom descrie în continuare arhitectura cîtorva calculatoare hipercub.

(x) *Cosmic Cube*

Calculatoarele Cosmic Cube de la California Institute of Technology (Seitz 1985) şi derivatul lui comercial, Intel iPSC (supercalculator personal) sînt ambele reţele hipercub de microprocesoare. Cosmic Cube este un hipercub 2^6 ce foloseşte un VAX 11/780 ca gazdă. Fiecare nod al cubului este constituit dintr-un Intel 8086 cu un coprocesor în virgulă mobilă Intel 8087, 128 KB RAM; plus 8 KB ROM ca memorie locală. Comunicaţia între procesoare se realizează prin cozi de aşteptare, pentru mesajele

transmise de-a lungul muchiilor cu o viteză de 2Mb/s. Fiecare procesor este conectat la 6 vecini. Există canale de comunicație asincrone independente full duplex, fiecare cu o coadă de așteptare pentru un mesaj de 64 biți. Se admite o durată a comunicației între noduri mare în comparație cu timpul de execuție al unei instrucțiuni, dar comparabilă cu timpul necesar nodului pentru interpretarea mesajului. Această arhitectură face sistemul un simulator convenabil pentru viitoarele mașini cu noduri pe un singur cip. În sens larg, mașina poate fi descrisă ca echivalentul a 64 IBM PC ce colaborează la rezolvarea unei probleme.

8087 are o performanță maximă de 50 Kflop/s, deci, pentru cele 64 procesoare ale calculatorului Cosmic Cube, s-ar atinge o viteză maximă de aproximativ 3 Mflop/s. De aproximativ 10 ori mai mult decît VAX 11/780 la aproximativ același cost. Totuși, mașina realizează numai 1/100 la 1/10 din viteza supercalculatoarelor mari. Mașina este operațională din octombrie 1983 ; o serie de probleme de fizică numerică au fost programate și executate cu succes pe ea. Un hipercub cu 2^{10} noduri, denumit Homogeneous Machine, ar urma să folosească Intel 80286 plus 80287, plus 80186 și o memorie locală de 256 KB (prin folosirea unor circuite de 64 Kb), expandabilă la 1 MB prin folosirea circuitelor de 256 Kb. Se estimează performanța hipercubului : 2^{10} la aproximativ 100 Mflop/s, aproximativ aceeași, cu a supercalculatoarelor mari (CRAY X-MP și CYBER 205).

(xi) *Intel iPSC-VX și N-Cube*

În cursul anului 1985, firma Intel a anunțat hipercuburle iPSC cu dimensiunile 2^5, 2^6 și 2^7 cu performanțe de 2, 4 și 8 Mflop/s. Fiecare nod procesor conține un microprocesor Intel 80286 cu un coprocesor aritmetic 80287 cu viteză de calcul de 1/16 Mflop/s. Se folosesc 512 KB RAM și 64 KB EPROM. Hipercubul cu 32 noduri ocupă un spațiu cu volumul de $0{,}6 \times 0{,}6 \times 2{,}4$ m^3 și este controlat de un microcalculator Intel 310. O altă companie, N-Cube, comercializează hipercuburi cu noduri circuite VLSI pe 32 biți special proiectate. Viteza este de 3 Mflop/s, iar memoria locală are 128 KB. Primul produs, N-Cube/10, poate fi extins la un cub 10-dimensional, cu 1024 noduri.

În 1986 Intel a anunțat iPSC-VX care are atașat fiecărui nod un procesor pipeline, ce crește performanța vectorială teoretică la 20 Mflop/s pe nod în modul de 32 biți și 6,7 Mflop/s în modul pe 64 biți. Deoarece un nod conține două plăci, numărul lor maxim se reduce la 64. Și cum viteza de comunicare între noduri nu s-a îmbunătățit, este probabil ca performanța acestei mașini să fie determinată în primul rînd de timpul consumat pentru comunicații, mai degrabă decît de cel pentru operații aritmetice (de exemplu, valoarea lui $f_{1/2}$ va fi mare, vezi §1.3.6). Din acest motiv, se impune o formulare foarte atentă a problemei și o programare adecvată.

(xii) *Seria FPS-T*

Tot în anul 1986, a anunțat și firma Floating Point Systems seria T, sau Tesseract, un hipercub ce poate fi extins, teoretic, la $2^{14} = 16384$ noduri. Cu toate acestea, primele mașini vîndute în 1986 Universității tehnice Michigan, Universității Grenoble și Laboratorului UK SERC

Daresbury au avut numai 16 sau 32 noduri. Fiecare nod poseda un transputer INMOS (vezi §3.5.5) pentru gestiunea comunicațiilor și 1 MB de memorie video DRAM biport ca memorie centrală. Memoria DRAM poate furniza un vector continuu de 256 elemente pe 32 biți în paralel la pînă la 4 registre vectoriale, care alimentează la rîndul lor 2 circuite aritmetice pipeline în virgulă mobilă pe 64 biți, WEITEK. La o perioadă a ceasului de 125 ns, s-ar atinge 16 Mflop/s pe nod. Se presupune că vectorii sînt memorați în spații continue din DRAM și că atît sumatorul, cît și multiplicatorul sînt folosiți simultan. Transputerul INMOS are rolul de a rearanja datele în formă continuă, simultan cu activitatea unităților pipeline. Raportul între timpii pentru execuția unei operații aritmetice, cel pentru rearanjarea datelor în DRAM și timpul necesar obținerii datelor de la un nod vecin este 1 : 26 : 256. Din nou, comunicația dintre noduri este cu cel mult 2 ordine de mărime mai lentă decît operatorul aritmetic, iar observațiile anterioare privind performanțele calculatorului Intel iISC se aplică, evident, și seriei T.

(xiii) *Denelcor HEP — MIMD pipeline*

Este interesant și probabil semnificativ că primul calculator MIMD comercializat, Denelcor HEP (Smith 1978). este destul de diferit față de calculatoarele descrise pînă acum. În loc de a avea un număr de unități de prelucrare a instrucțiunilor distincte, fiecare cu fluxul propriu de instrucțiuni, la HEP întîlnim mai multe fluxuri de instrucțiuni ce partajează o singură unitate de prelucrare a instrucțiunilor, pipeline. Arhitectura generală a unui sistem HEP-heterogeneous element processor-complet constă din 16 module de execuție a proceselor (PEM) conectate la 128 module de memorie pentru date (DMM) printr-o rețea de comutare a pachetelor. Fiecare PEM conține 1 Mcuv. memorie pentru programe, 2 Kcuv. memorie pentru registre și 4 Kcuv. de memorie pentru constante. Fluxurile de instrucțiuni se creează sub controlul programului : instrucțiunea mașină (sau comanda FORTRAN) CREATE inițiază un flux nou de instrucțiuni, numărul maxim de fluxuri ce pot coexista fiind de 50. Apoi, instrucțiunile fiecărui flux sînt transmise unui pipeline cu 8 etaje. Cînd acest pipeline este plin, o instrucțiune este executată la fiecare 100 ns., de unde atingerea unei viteze maxime de 10 Mips pe PEM sau 160 Mips pentru un sistem HEP complet. Dacă presupunem că în medie pentru fiecare operație în virgulă mobilă sînt necesare 5 instrucțiuni, se obține 2 Mflop/s pe PEM și 32 Mflop/s pentru întregul sistem. Dacă numărul instrucțixnilor pentru o operație în virgulă mobilă se poate reduce, viteza poate fi crescută la 5 sau 6 Mflop/s pe PEM. Aceasta se poate realiza pentru anumite calcule cu matrici, dacă registrele PEM sînt programate să lucreze ca registre vectoriale ce memorează rezultatele intermediare. În acest mod se limitează numărul de referințe la memoria de date (*Sorensen 1984*, *Dongarra și Sorensen 1985*).

Metoda implementării calculatoarelor MIMD cu PEM, în cazul HEP, este mult mai flexibilă decît cea folosită la structurile ce se bazează pe un număr fix de microcalculatoare. Numărul fluxurilor de instrucțiuni poate fi schimbat de la o problemă la alta prin program și, în acest mod, se poate alege un număr adecvat de fluxuri pentru problema de rezolvat — nu mai este un număr fixat prin hardware, ce s-ar putea să nu corespundă

problemei sau algoritmului ce este implementat. Fiecare DMM poate avea pînă la 1 Mcuv. de 64 biţi, ceea ce înseamnă o capacitate de 1 GB pentru sistemul complet. Reţeaua de comutare asigură un timp de propagare între noduri de 50 ns. Un interval de timp tipic pentru a obţine o dată din memorie, via reţea, este de 2 microsec. Datorită arhitecturii sale interesante, vom trata despre Denelcor HEP în § 3.4.

Dezvoltarea sistemului HEP a fost subvenţionată de armata S.U.A. şi a culminat cu livrarea unui sistem cu 4 PEM×4 DMM la Ballistic Research Laboratories, Aberdeen-Maryland, în anul 1982. Este interesant că tot BRL a subvenţionat şi achiziţionat în 1946 calculatorul ENIAC, care poate fi considerat primul calculator MIMD. Sisteme cu un singur PEM/DMM au fost livrate la University of Georgia Research Foundation (1982), Messerchmidt Research Munich (1983) şi Los Alamos Research Laboratories (1983). Versiunea iniţială, descrisă mai sus ca HEP-1, a fost implementată în tehnologie ECL 10K, deoarece nu s-a dorit lansarea în acelaşi timp cu arhitectura revoluţionară şi a unei noi tehnologii. HEP-2, anunţat în 1983 pentru livrarea în 1986, urma să folosească tehnologia VLSI ECL îmbunătăţită, cu un timp de comutare de 300 ps şi 2500 porţi/cip. Perioada ceasului ar trebui să fie de 20 ns, iar o unitate pipeline pentru înmulţire/adunare ar realiza 100 Mflop/s pe PEM. El se bazează pe arhitectura MIMD a lui HEP-1 şi va atinge o performanţă de între 200 Mips şi 12000 Mips, ce corespund în mare la valorile de 50 Mflop/s şi 2,4 Gflop/s. Din păcate, compania a dat faliment în 1985 datorită problemelor financiare. Totuşi, proiectul HEP este atît de nou şi interesant încît îl vom trata pe larg în § 3.4.4.

(xiv) *CDC Cyber Plus — un sistem în inel*

Calculatorul Cyber Plus, produs de Control Data, este derivat din Advanced Flexible Processor, construit pentru aplicaţii militare ca analiza rapidă a fotografiilor luate din avion. Arhitectura lui Cyber Plus este neobişnuită şi interesantă pentru că se bazează pe o topologie de comunicaţii multi-inel.

Sistemul CDC Cyber Plus este format din 1 pînă la 16 procesoare Cyber Plus conectate în inel la un canal al unui CDC Cyber 170/800, folosit ca gazdă. În clasificarea noastră (vezi § 1.2.6) este un calculator MIMD cu reţea în inel. La calculatorul gazdă pot fi conectate pînă la 4 astfel de inele. Procesorul propriu-zis, Cyber Plus, posedă 256 sau 512 Kcuv. de 64 biţi (timp de acces 80 ns) pentru date în virgulă mobilă, 16 Kcuv. de 16 biţi (timp de acces 20 ns) pentru întregi şi o memorie program de 4 Kcuv. iar instrucţiunile pe 240 biţi definesc destinaţia rezultatelor produse de fiecare unitate funcţională, la fiecare ciclu maşină.

Procesoarele Cyber Plus formează staţii pe un inel, iar comunicaţiile între procesoare se realizează prin trimiterea unor pachete de informaţii pe inel. Pachetele se deplasează de-a lungul inelului la viteza de o staţie într-o perioadă de ceas, pînă ce este atinsă destinaţia. Fiecare procesor poate citi sau scrie pe inel la fiecare perioadă de ceas. Deci, timpul de comunicaţie între procesoare depinde de distanţa relativă în inel. Un al doilea inel leagă procesoarele Cyber Plus la gazda Cyber 170 şi memoria sa.

(xv) *Sisteme conectate prin magistrala (ELX SI 6400, FPS 5000, Sequent Balance)*

De departe, calculatoarele MIMD comercializate cele mai comune folosesc mai multe elemente de calcul (CE) și module de memorie conectate la o magistrală comună. În continuare vom descrie unele dintre ele. Alte calculatoare, cu o arhitectură similară, sînt FLEX/32 (20/cabinet), Encore Multimax (20) și Culler PSC (2); în paranteză am trecut numărul de CE.

(xvi) *ELXSI 6400*

ELSXI 6400 este un exemplu de sistem MIMD în care procesoarele și memoria sînt conectate la o magistrală rapidă partajată. La ELXSI (Taylor 1983) pot exista de la 1 la 10 CPU și de la 1 la 10 procesoare de I/E ce accesează de la 1 la 6 sisteme de memorie (4 la 192 MB), via o magistrală de date comună de mare viteză, denumită Gigabus. Unitățile centrale pe 64 biți ating 4 Mips, astfel că viteza potențială maximă este de 40 Mips. Gigabus are lărgimea de 64 biți și un ciclu de 25 ns, asigurînd o viteză maximă de transfer de 320 MB/s (adrese și date). În mod obișnuit se atinge de la 160 MB/s la 220 MB/s. Ciclul memoriei este de 400 ns (pentru două citiri pe 64 biți și o scriere), iar timpul de înmulțire în virgulă mobilă este de 300 ns (64 biți). Luînd de asemenea în considerare ciclurile de magistrală necesare, se obține un timp total pentru execuția unei înmulțiri în virgulă mobilă cu 1 CPU de aproximativ 800 ns, deci o viteză de 1,2 Mflop/s. Programele Livermore se execută în mod curent la 0,3 pînă la 1,4 Mflop/s pe un sistem cu 1 CPU. Prin suprapunerea acceselor la memorie pentru operațiile aritmetice, un sistem de memorie poate asigura cu date 2 CPU fără interferențe, deci 6 sisteme de memorie sînt mai mult decît suficiente pentru sistemul complet cu 10 CPU, ce ar atinge astfel 12 Mflop/s. Un calculator ELXSI 6400 a fost comandat de către NASA Ames Dryden Data Analysis Facility.

(xvii) *FPS-5000*

Seria de calculatoare Floating Point Systems 5000 este, ca și ELXSI, un sistem MIMD cu comunicații prin magistrale (Cannon 1983). Un procesor de control de 8 sau 16 Mflop/s și pînă la 3 coprocesoare aritmetice XP 32 de 18 Mflop/s sînt conectate via o magistrală de 6 Mcuv./s la o memorie comună de 1 Mcuv. Procesorul de control este fie un FPS AP-120B, fie un FPS-100 „masiv de procesoare”, dar XP 32 este un proiect nou ce folosește circuitele VLSI WEITEK pentru înmulțire și adunare pe 32 biți în virgulă mobilă. Acestea sînt pipeline-uri cu 8 etaje ce lucrează cu o perioadă a ceasului de 167 ns, de unde viteza de 6 Mflop/s pe circuit. Un circuit pentru înmulțire furnizează rezultatele la 2 circuite pentru adunare, într-o manieră convenabilă calculului transformatei Fourier rapide, asigurîndu-se 18 Mflop/s pe coprocesor și 62 Mflop/s pentru configurația maximă a sistemului. Procesorul de control și coprocesoarele pot executa programe independente. FPS-5000 este descris mai pe larg în § 2.5.10.

(xviii) *Sequent Balance 8000, 21000*

Proiectat ca un superminicalculator cu performanțe de aproximativ 6 ori mai mari ca ale lui DEC VAX 11/780, Sequent Balance este o arhitectură pe 32 biți, cu o magistrală pentru conectare la memoria comună.

Pînă la 12 PE (modelul 8000) sau 24 PE (modelul 21000) şi pînă la 28 MB de memorie comună partajabilă se pot ataşa la o magistrală largă de 52 biţi cu transfer pipeline de 26,7 MB/s. Fiecare proces poate folosi un spaţiu de adresare virtuală de 16 MB. Fiecare procesor este un NS 32032 ce operează la 10 MHz, are unităţi în virgulă mobilă, unitate de management a memoriei şi 8 KB de memorie cache. Pe o placă se află 2 PE, ce comunică cu magistrala printr-un circuit propriu care asigură comunicaţia inter-procesoare, sincronizarea şi gestiunea întreruperilor. Aceste sisteme au fost introduse în producţie în 1984, iar la sfîrşitul lui 1985 se vînduseră aproximativ 80 bucăţi. Acest calculator este popular, permiţînd într-un mod economic investigarea algoritmilor paraleli pentru sistemele cu memorie partajată. În acest scop este folosit în cadrul Departamentului de calculatoare al Universităţii tehnologice Loughborough.

(xix) *IBM/CAP şi Cornell Supercomputer Center*

Relaţia dragoste-ură a IBM-ului cu domeniul supercalculatoarelor ştiinţifice, deja amintit, a căpătat o turnură pozitivă la începutul anilor 80 sub conducerea Dr. E. Clementi, devenit IBM Fellow, angajat în cercetări de chimie numerică cu ajutorul firmei IBM. Fiind dezamăgit de facilităţile de calcul oferite de produsele IBM, Clementi a conceput ideea ataşării a pînă la 10 calculatoare Floating Point Systems 164 la canalele unui calculator gazdă IBM. Calculatoarele formează o reţea în stea cu centrul calculatorului gazdă, între cele 10 calculatoare FPS 164 nefiind nici o legătură directă. Deşi rata de transfer a canalului este lentă în comparaţie cu timpul de execuţie al operaţiilor de calcul, Clementi şi colegii săi au găsit că multe probleme ştiinţifice pot fi împărţite în sub-task-uri care nu necesită comunicaţii prea frecvente prin canal. Se asigură că acest masiv de procesoare slab interconectate (loosely coupled array of processors — ICAP) poate atinge viteze de calcul comparabile cu cele ale lui CRAY-1, la un cost foarte economic. ICAP este descris în detaliu în § 2.5.9:

Primul sistem ICAP a fost instalat la IBM Kingston, iar al doilea în 1985/6 la IBM European Center for Scientific and Engineering Computing (ECSEC) Roma, pentru a fi folosit de oamenii de ştiinţă europeni. Un sistem similar se află la Universitatea Cornell, ca unul din cele 4 centre de calcul avansat stabilite de către US National Science Foundation. La aceasta a colaborat şi Cornell Center for Theory and Simulation în Science and Engineering, condus de K.G. Wilson, laureatul Nobel pentru fizică în anul 1982.

Configuraţia iniţială care folosea 10 FPS 164 pentru a atinge viteza lui CRAY-1 nu este, probabil, prea atrăgătoare, dar experienţa cîştigată arată cum se pot împărţi problemele mari pentru a fi executate de astfel de sisteme MIMD. Oricum, sînt iminente trei dezvoltări, care pot transforma această imagine. Prima, o magistrală de mare viteză, în curs de realizare, va permite calculatoarelor FPS 164 să comunice între ele direct, fără a mai folosi canalele lente ale calculatorului gazdă; a doua, FPS 164 pot fi înlocuite cu FPS 264 cu o creştere de 4—5 ori a vitezei cu păstrarea compatibilităţii programelor; a treia, la fiecare FPS pot fi adăugate pînă la 15 plăci FPS MAX, ceea ce ar conduce la o performanţă teoretică maximă de 341 Mflop/s pentru fiecare FPS şi 3,4 Gflop/s pentru configuraţia ICAP completă.

(xx) *BBN Butterfly*

Acest calculator diferă de celelalte sisteme MIMD descrise prin aceea că este un sistem cu memorie distribuită (vezi §1.2.6 şi fig. 1.9). Cu alte cuvinte, întreaga memorie a sistemului este distribuită ca memorie locală multiplelor PE, interconectate într-o reţea cu mai multe etaje. Natura distribuită a memoriei nu influenţează modul de programare, utilizatorii avînd posibilitatea să scrie programe ca şi cum întreaga memorie este partajată de toate CE. Singura diferenţă între accesarea datelor în memoria locală a CE, şi în memoria locală a altui CE constă în timpul necesar mesajelor să parcurgă reţeaua. Acesta este de 4 microsec., ceea ce este foarte mult pentru un sistem de calcul. Topologia reţelei este Banyan (Feng 1981), similară celei necesare pentru furnizarea datelor trasformatei Fourier rapide în $\log_2 n$ paşi pentru n date (aşa numita operaţie fluture, vezi de asemenea §3.2.2. şi §5.5.1). De aici şi denumirea reţelei.

Primul model BBN Butterfly (1985) conţine 128 CE, cu posibilitatea maximă de 256, ca CE folosindu-se Motorola 68000 cu 1 MB de memorie locală. Îmbunătăţiri ulterioare includ operatori în virgulă mobilă hardware, cu Motorola 68020 şi coprocesorul 68881, şi o memorie locală de 4 MB. Pe o placă se află 1 CE cu memoria sa. Calculatorul iniţial Butterfly cu 128 CE a atins 26 Mflop/s la înmulţirea a două matrici 400×400 şi 3 Mflop/s la rezolvarea a 1200 ecuaţii liniare prin eliminare gaussiană.

(xxi) *Transputerul INMOS*

O etapă importantă în dezvoltarea reţelelor MIMD de calculatoare a fost reprezentată de anunţarea în anul 1985 a Transputerului INMOS (INMOS 1985). Acesta este o familie de circuite VLSI proiectate special pentru lucrul concurent şi limbajul de programare paralelă OCCAM asociat. De exemplu, transputerul T 800 constă, pe un cip de 1 cm^2., dintr-un microprocesor pe 32 biţi, un coprocesor în virgulă mobilă, 4 Kb de memorie şi 4 legături seriale pe 1 bit pentru conectarea cu alte transputere într-o reţea.

Acum se află în curs de realizare mai multe calculatoare MIMD ce folosesc transputere şi, în mod evident, transputerul va avea o influenţă semnificativă asupra reţelelor MIMD. De aceea descriem transputerul în capitolul 3 (§3.5.5), iar limbajul OCCAM în capitolul 4 (§4.4.2).

(xxii) *Circuitele VLSI în virgulă mobilă WEITEK*

O altă realizare VLSI cu o mare influenţă asupra implementării şi, în particular, a performanţelor calculatoarelor MIMD. o reprezintă circuitele VLSI pipeline pentru operaţii aritmetice în virgulă mobilă. Înaintea anului 1984, un sumator în virgulă mobilă pe 64 biţi necesita aproximativ 13000 porţi, implementate în tehnologie de integrare pe scară medie (MSI, cu 10 pînă la 100 porţi/cip). Se foloseau aproximativ 2000 circuite plasate pe 7 plăci de 35 cm.$\times$54 cm. (Charlesworth şi Gustafson 1986). În cursul anului 1984 firma WEITEK Inc., din Valea Siliciului, California, a realizat trecerea la integrarea pe scară foarte largă (VLSI cu mai mult de 1000 porţi/cip) şi a oferit posibilitatea de a executa operaţii în virgulă mobilă pe 64 biţi într-o unitate pipeline realizată cu 9 circuite VLSI (Ware et al 1984). La mijlocul anului 1985 şi alţi producători au oferit produse similare (de exemplu, Analog Devices). Circuitele WEITEK pe 64 biţi

includ un multiplicator în 7 etaje și un sumator cu 6 etaje, ce lucrează la o perioadă a ceasului de 125 ns. Deci, performanța teoretică maximă este de 16 Mflop/s pentru numai două circuite VLSI. Este deci posibil pentru fiecare proiectant să obțină o performanță teoretică maximă de 16 Mflop/s pe CE (sau 1 Gflop/s pentru sisteme cu 64 CE) prin folosirea acestor circuite în cadrul fiecărui CE al unui sistem MIMD. Desigur, fracția din această performanță teoretică ce se poate obține în mod real poate fi foarte mică dacă comunicațiile nu se execută în mod adecvat, pentru a alimenta în permanență circuitele cu date. Nu este mai puțin adevărat că potențialul aritmetic al sistemului MIMD a fost revoluționat de aceste componente care se pot întîlni acum în cadrul multor sisteme (Alliant FX/8, FPS T-Series, FPS-5000, FPS-164/MAX).

1.1.9. Minisupercalculatoarele (Convex C-1, SCS-40)

Denumite uneori „supercalculatoarele accesibile", (ca preț, nota trad.) minisupercalculatoarele se pot defini ca sisteme de calcul vectoriale în virgulă mobilă pe 64 biți cu de la 1/8 la 1/4 din performanța unui supercalculator, dar la aproximativ 1/10 din preț, deci la prețul unui minicalculator. De exemplu, Convex C-1, anunțat în 1984, atinge maximum 20 Mflop/s (64 biți) sau 40 Mflop/s (32 biți) pentru un preț de 495 000$ (Linback 1984). (În tab. 1.1 se compară performanțele acestui minisupercalculator cu ale altora). Apariția acestui calculator este un alt aspect al revoluției calculatoarelor produsă de circuitele VLSI cu 10000 sau mai multe porți pe cip. Nu numai că această tehnologie permite ca un calculator serial simplu să fie implementat pe un singur cip, obținînd microprocesorul, dar întreaga logică a unui calculator vectorial pipeline poate fi plasată pe cîteva sute de circuite care pot ocupa numai cîteva plăci. Ca un exemplu extrem, ETA Corporation a demonstrat că un CYBER 205 cu 2 unități pipeline poate fi realizat pe o placă de 48 cm × 60 cm, folosind circuite VLSI CMOS cu 20000 porți/cip (vezi §2.3.7.).

(i) *Convex C-1*

Convex C-1 folosește 8000 porți/cip, de aceea mașina cu 128 MB de memorie ocupă numai un rack de 48 cm și 1,5 m înălțime. Un al doilea rack conține o unitate de bandă și una de disc. La o putere consumată de 3,2 KW, răcirea cu aer este suficientă și, deci, nu necesită condiții speciale de instalare. Spre deosebire de acesta CRAY X-MP folosește cipuri cu 16 porți, consumă aproximativ 200 KW și are nevoie de o instalație specială de răcire cu freon, de unde și o cauză a diferenței de preț. Vitezele relative de calcul trebuie să fie în linii mari invers proporționale cu perioadele ceasului care sînt de 12,5 ns pentru CRAY-1 și 100 ns pentru C-1, de unde un raport de 8, verificat practic.

Performanța pentru probleme reale poate fi judecată și stabilită în contextul bibliotecii de programe de test LINPACK (Dongarra 1986) unde 300 ecuații liniare sînt rezolvate în FORTRAN prin folosirea tehnicii matrice-vector (vezi tabelul 1.1). Performanțele relative sînt de 66 Mflop/s pentru CRAY-1S, 8,7 Mflop/s pentru Convex C-1 și 0,1 Mflop/s pentru DEC VAX 11/780 cu accelerator în virgulă mobilă (folosit pe scară largă

în anul 1984). Optimizarea execuţiei de programator pentru codurile critice creşte performanţa lui C-1 la 14 Mflop/s. Pe de altă parte, un supercalculator al anului 1984, CRAY X-MP/4, ce foloseşte toate cele 4 CPU, poate atinge 480 Mflop/s pentru această problemă.

Tabelul 1.1 **Comparaţii între minisupercalculatoar, prin execuţia unor programe de test. Performanţa medie este dată în Mflop/s pentru problema dată în operaţii în virgulă mobilă pe 64 biţi (excepţiile sînt menţionate). Valorile comparabile pentru cele mai performante supercalculatoare sînt prezentate în tabelul 2.7 (Date din lucrările lui Dongarr 1985, Dongarra şi Sorensen 1985, 1987).**

Problema	MINI VAX[b] 11/780FIA	MINISUPER Alliant FX/(p)	MINISUPER Convex C-1	MINISUPER SCS 40	SUPER CRAY X-MP/(p)
Performanţa		5,9(1)	20	44	210(1)
teoretică maximă		47(8)	40(32biţi)		420(2)
					(840)4
(r, n_{12})		(0,9,151)(1			(70,53) (1)
FORTRAN 1.3.3		(1,1, 23)(8)			(140,5700) (2)
Livermore 3					96(1)
produsul cel mai interior					
Livermore 6					8(1)
tridiagonal					
Livermore 14					7(1)
analiza particulelor					
LINPACKa		1,3(1)	2,9		24(1)
FORTRAN		2,5(8)		7,3	
n = 100		6,2(8)			
Asamblor		1,7(1)	3,2		44(1)
bucla cea mai interioară	2,6(8)				
n = 100		8,5(8)			
Asamblorul cel mai bun					171(1)
matrice-vector	0,11	7,3(8)	8,7		257(2)
n = 300		14(8)	14	26	480(4)

Note

[a] Soluţia a n ecuaţii liniare (Dongarra et al. 1979)
[b] Minicalculator tipic pe 32 biţi, la nivelul anului 1984
[c] Toate FORTRAN
[d] Numărul de CE sau CPU ofolosit este trecut în paranteze
[e] FORTRAN cu directive pentru compilator
[f] Hockney (1985 a)

Semnificaţia apariţiei minisupercalculatoarelor este că firmele de inginerie ce foloseau VAX 11/780 sau minicalculatoare similare pentru calcule tehnice îşi pot creşte posibilităţile de calcul cu aproximativ 2 ordine de mărime prin introducerea unui supercalculator, fără o creştere semnificativă a preţului. Deci, simulările inginereşti complexe, executate anterior de centre dotate cu supercalculatoare, pot fi executate acum „acasă''. Mai mult calitatea operaţiilor aritmetice a crescut de la 32 biţi pentru VAX la 64 biţi pentru minisupercalculatoare. Totuşi, deşi Convex

C-1 realiza în mod real în anul 1984 între 1/8 și 1/4 din performanța supercalculatoarelor anului 1976 (CRAY-1), rezultatele execuției programelor LINPACK arată, de asemenea, că se realizează numai 1/50 din performanța unui supercalculator al anului 1984 (CRAY X-MP/4).

În general, arhitectura lui Convex C-1 este similară celei a lui CRAY-1 în aceea că folosește o serie de unități funcționale ce lucrează cu registre vectoriale. Detaliile sînt totuși destul de diferite; mașina nu folosește setul de instrucțiuni al lui CRAY. Spre deosebire de CRAY-1, setul de instrucțiuni ale lui C-1 permite adresarea la nivel de octet, iar adresele pe 32 biți (în comparație cu 24 biți la CRAY-1) pot adresa direct un spațiu de memorie virtuală de 4 GB, sau 500 Mcuv. (64 biți). Există trei unități funcționale (pentru load/store/vector edit, adunare/operații logice și înmulțire/împărțire) și 8 registre vectoriale a 128 elemente de 64 biți fiecare. Fiecare unitate funcțională posedă 2 unități pipeline, pentru elemente pare, respectiv impare. Fiecare pipeline execută o operație pe 64 biți la fiecare 200 ns sau o operație pe 32 biți la fiecare 100 ns. Se obține o rată efectivă de prelucrare de 1 rezultat pe 64 biți la fiecare 100 ns, sau un rezultat pe 32 biți la fiecare 50 ns. Deoarece numai 2 unități pipeline execută operații aritmetice în virgulă mobilă, se obțin 20 Mflop/s pentru modul în 64 biți, sau 40 Mflop/s pentru modul în 32 biți. Registrele vectoriale primesc date fie de la o memorie cache de 64 KB cu timp de acces de 50 ns, fie direct de la memoria centrală de 16 Mcuv. organizată în 16 blocuri. Transferurile pe 64 biți între memoria centrală și memoria cache au loc la 10 Mcuv./s, în comparație cu 80 Mcuv./s pentru CRAY-1 și 315 Mcuv./s la CRAY X-MP.

(ii) *SCS-40*

Al doilea minisupercalculator anunțat, Scientific Computer System SCS-40, a apărut în 1986. Ca și în cazul lui Convex C-1, producătorii pretind că SCS-40 atinge 25% din performanța lui CRAY X-MP/1, la 15% din preț. Spre deosebire de C-1, această mașină folosește setul de instrucțiuni al lui CRAY, iar programele CRAY pot fi executate fără probleme. Arhitectura constă din 16 blocuri de memorie centrală (4 Mcuv. în configurație maximă) conectată via magistrale multiple și o rețea de comutare vectorială la 8 registre vectoriale a 64 elemente. Acestea, la rîndul lor, sînt conectate prin mai multe magistrale la unități funcționale pipeline, ce corespund celor de la seria CRAY. Fiecare unitate funcțională are o performanță maximă de un rezultat la fiecare tact de 45 ns, atingînd maximum 44 Mflop/s dacă cele două pipeline pentru adunare și înmulțire lucrează simultan.

Ciclul magistralei de 22,5 ns permite ca pe o singură magistrală să se transmită 2 cuvinte de 64 biți în cursul unei perioade de ceas a mașinii și deci, lucrează ca 2 magistrale logice. Între memoria centrală și registrele vectoriale operează 4 astfel de magistrale logice ce asigură o rată de transfer de 89 Mcuv./s (în comparație cu 10 Mcuv./s la C-1). Cele 6 magistrale logice asigurate între registrele vectoriale și unitățile funcționale permit execuția simultană a 2 operații diadice. Pentru a asigura aceste viteze de lucru s-au folosit circuite integrate MSI și LSI în tehnologie ECL, spre deosebire de circuitele VLSI CMOS mai lente de la Convex C-1 (perioada ceasului de 100 ns). Diferența de viteză este evidentă la execuția progra-

melor LINPACK pentru matrici de ordinul 100, cînd SCS-40 atinge 7,3 Mflop/s, iar Convex C-1 2,9 Mflop/s ; un raport aproape identic cu cel între perioadele de ceas ale celor 2 mașini. Reprogramarea problemelor LINPACK în termeni de matrice-vector, în loc de vector-vector ridică performanța de 26 Mflop/s pentru o matrice de ordinul 300.

(iii) *Alliant FX/8*

Alliant FX/8 este singurul minisupercalculator descris aici care folosește o arhitectură MIMD ; 8 elemente de calcul (CE) partajează o memorie comună. Fiecare CE este un calculator vectorial cu 8 registre vectoriale ce păstrează 32 cuvinte a 64 biți fiecare și unități funcționale pipeline separate pentru adunare, înmulțire și împărțire în virgulă mobilă. La un ciclu de 170 ns, fiecare CE atinge maximum 11,8 Mflop/s (32 biți) sau 5,9 Mflop/s (64 biți), deci o mașină cu 8 CE atinge maximum 94 Mflop/s (32 biți) sau 47 Mflop/s (64 biți). CE sînt conectate printr-o rețea cu accesuri încrucișate la 2 memorii cache de 64 KB la o rată de transfer de 376 MB/s. Memoria cache accesează memoria partajată printr-o magistrală la o rată de 188 MB/s. Memoria partajată se poate extinde la 64 MB în module de 8 MB, fiecare împărțire în 4 blocuri. În tabelul 1.1 se prezintă și rezultatele execuției programelor de test pe Alliant FX/8. Se observă că performanțele sînt similare lui Convex C-1 și inferioare celor ale lui SCS-40.

Mașina mai conține o „magistrală concurentă" direct conectată la toate CE. Aceasta este folosită pentru sincronizarea CE cu un overhead minim. Fiecare CE are o unitate de control concurent (CCU) care este interfața la această magistrală. CCU distribuie în timpul execuției operațiile la CE și sincronizează calculele prin hardware. În acest mod, se asigură prin hardware dependența de date a diferitelor instanțe ale buclelor DO, fără nici o intervenție a programatorului, chiar dacă la diferite CE se atribuie indici diferiți ai buclelor. Alliant FX/8 va fi folosit ca un ciorchine cu 8 PE în proiectul Cedar, deja amintit (§1.1.8). Este de asemenea vîndut separat, ca stație de lucru Apollo Domain. Primele livrări s-au efectuat în 1985. Modulul cu 1 CE este cunoscut ca Alliant FX/1. Acesta are memorie cache de 32 KB și 1 sau 2 module de memorie de 8 MB.

1.2. Clasificarea arhitecturilor

Am văzut pe parcursul prezentării istoriei paralelismului că au fost propuse un număr mare de arhitecturii paralele diferite, din care o parte au fost realizate, chiar dacă numai în forma experimentală. Încercările de a clasifica aceste sisteme nu au avut un succes unanim recunoscut și nu există încă (c 1987) o schemă de clasificare sau notație general acceptate. Vom prezenta, totuși, în §1.2.2 taxonomia lui Flynn (1972) și în §1.2.3 pe cea a lui Shore (1973) deoarece amîndouă au fost discutate pe larg, iar unii termeni au fost adoptați de limbajul științei calculatoarelor. Deficiențele acestor clasificări constau în aceea că unele arhitecturi bine cunoscute, în special cele pipeline nu aparțin în mod clar unei clase, iar altele, ca ICL DAP, pot fi repartizate la fel unor clase diferite. O altă posibilitate este de a analiza modalitățile principale în care intervine parale-

lismul în arhitectura calculatoarelor actuale: pipelining, multiplicarea procesoarelor, paralelismul funcţional. Astfel, se realizează o apropiere mai mare de realitate, dînd naştere unei taxonomii mai uşor de aplicat decît conceptele teoretice ale lui Flynn şi Shore.

Oricum, prima fază în definirea unei clasificări realiste este descrierea cu acurateţe şi concizie a caracteristicilor esenţiale ale arhitecturilor analizate, iar aceasta necesită o notaţie adecvată. De aceea, vom prezenta în paragraful §1.2.4 o notaţie care permite descrierea unei arhitecturi cu o singură linie de text. Este o metodă apropiată de exprimarea formulelor chimice pentru moleculele mari. Această notaţie este apoi folosită în §1.2.5 pentru clasificarea structurală a calculatoarelor seriale şi paralele prezentate în această carte.

1.2.1. Nivele de paralelism

Istoria arată că paralelismul a fost introdus încă de la început pentru a creşte eficienţa utilizării calculatoarelor, iar aceasta s-a aplicat la nivele distincte care pot fi clasificate astfel:

1. *Nivelul job-ului*
   (i) între job-uri
   (ii) între fazele unui job.
2. *Nivelul programului*
   (i) între părţi ale programului;
   (ii) în interiorul buclelor DO;
3. *Nivelul instrucţiunii*
   (i) între fazele de execuţie ale instrucţiunii.
4. *Nivelul aritmetic şi al bit-ului*
   (i) între elementele unei operaţii vectoriale;
   (ii) în interiorul circuitelor logice aritmetice.

La nivelul cel mai înalt, instalarea unui calculator are ca obiectiv maximizarea vitezei de execuţie a job-urilor. Execuţia unui job poate fi împărţită mai multe faze secvenţiale, fiecare avînd nevoie de anumite programe şi anumite resurse hard ale sistemului. Fazele tipice pot fi: citirea codului sursă FORTRAN; compilarea; editarea legăturilor; execuţia; tipărirea rezultatelor. Deoarece operaţiile de I/E sînt mai lente decît unitatea centrală, se introduc mai multe canale de I/E sau procesoare de periferice, care pot opera în paralel cu execuţia programului. Cele mai multe sisteme au un singur procesor pentru execuţia programelor, dar unele au două sau mai multe. *Sistemul de operare* are sarcina de a organiza distribuirea resurselor pentru diferitele job-uri. De obicei sînt rezidente în memoria centrală a calculatorului mai multe programe (5—10), iar în cazul unui procesor, la un moment dat, se execută un singur program. Imediat ce programul solicită o operaţie de I/E care este lentă, aceasta este transmisă canalului I/E şi în execuţia unităţii centrale intră un alt program. Primul program aşteaptă disponibilitatea tuturor datelor şi primeşte din nou controlul cînd celelalte programe sînt şi ele obligate să aştepte. Astfel, operaţiile de I/E ale unui job se suprapun cu execuţia altui

job, într-o manieră dinamică funcţie de necesităţile job-urilor ce partajează resursele calculatorului la un moment dat.

Astfel de sisteme de operare nu cunosc legăturile logice interne ale programului executat şi de aceea trebuie să urmărească execuţia secvenţială a diferitelor faze ale job-ului, ca de exemplu terminarea operaţiilor de I/E înainte de a trece la execuţia unei noi instrucţiuni. În unele cazuri programatorul, poate controla operaţiile de I/E şi planifica transferurile de date în blocuri. În aceste cazuri el poate folosi zone tampon pentru a suprapune operaţiile de I/E cu execuţia programului. Pentru zone tampon cu 3 nivele, de exemplu, se folosesc 2 canale iar în zonele tampon sînt stocate 3 blocuri de date. Simultan este citit în zona tampon un bloc nou, via canalul de intrare, datele din zona tampon 2 sînt prelucrate de procesor, iar ultimul bloc de valori calculate este transmis canalului de ieşire via zona tampon 3. Calculele continuă prin modificarea ciclică a rolurilor zonelor tampon.

Din cele de mai sus se observă că cerinţa principală a paralelismului la nivelul job-ului impune asigurarea unui număr adecvat de resurse multiplicate, ceea ce este clasificat cu denumirea de paralelism funcţional. De aceea, este important de a identifica strangulările în activitatea sistemului pentru a suplimenta (sau reduce) resurse funcţie de circumstanţe.

Vom considera în continuare acel tip de paralelism ce intervine în execuţia unei faze dintr-un job. În interiorul unui program pot exista părţi de cod care sînt relativ independente una faţă de alta şi pot fi executate în paralel de mai multe procesoare (de un set de microprocesoare, de exemplu). Unele secţiuni de cod independente pot fi recunoscute printr-o analiză logică a codului sursă, dar altele depind de date şi astfel nu sînt cunoscute pînă în faza de execuţie. În alt caz, execuţiile diferite ale unei bucle pot fi independente una de alta, funcţie de instrucţiunile condiţionale ale buclei. Fiecare microprocesor primeşte codul complet şi se pot executa în paralel atîtea trasee în buclă cîte microprocesoare sînt. Această situaţie apare la metoda Monte-Carlo şi are aplicaţii importante în ingineria nucleară. Problemele de programare asociate cu astfel de structuri de microprocesoare sînt un subiect de cercetare, şi multe sisteme au devenit operaţionale în anii '80 (vezi §1.1.8 şi §1.2.6).

Toate companiile care produc calculatoare pentru operaţii cu vectori (Burroughs, TI, CRAY) au realizat compilatoare FORTRAN care recunosc cînd o buclă DO poate fi înlocuită cu una sau mai multe instrucţiuni vectoriale. Se recunosc, astfel buclele DO care reprezintă scalar un set de operaţii vectoriale ce pot fi executate mai eficient cu instrucţiuni vectoriale. Principalele arhitecturi folosite sînt unităţile aritmetice pipeline pentru calculatoarele vectoriale pipeline (ex. CYBER 205, CRAY-1) sau multiplicarea elementelor procesoare în cazul masivelor de procesoare (ex. ICL DAP şi BSP).

La nivel inferior am menţionat deja că execuţia unei instrucţiuni poate fi împărţită în mai multe suboperaţii, iar principiul pipeline poate fi folosit pentru suprapunerea diferitelor suboperaţii ale diferitelor instrucţiuni. Această metodă este aplicată pe scară largă la procesoarele scalare rapide (ex. ICL 2900, AMDAHL 470 V/6) şi unele masive de procesoare (ex. BSP). O parte din execuţia unei instrucţiuni este calculul aritmetic şi aceasta poate fi împărţită la fel în suboperaţii ce se pot executa într-un pipeline.

La nivelul cel mai de jos, se poate interveni în logica aritmetică realizînd operații în mod serial pe bit sau pe toți biții unui număr în paralel. Există și posibilități intermediare : de exemplu, efectuarea operațiilor în paralel cu biții unui octet, dar cu octeții secvențial. Această metodă a fost intens studiată la prima generație de calculatoare a anilor '50. După ce execuția operațiilor în virgulă mobilă paralelă pe bit a devenit standard prin 1955, acest nivel de paralelism a fost uitat. Apariția în anii '70 a microprocesoarelor a actualizat din nou metoda prezentată. Mai mult, dorința de a realiza masive mari de procesoare la un preț rezonabil a impus execuția operațiilor aritmetice serial pe bit, ca la STARAN și ICL DAP.

1.2.2. Taxonomia lui Flynn

Flynn nu și-a bazat clasificarea sa macroscopică a arhitecturilor paralele pe structura mașinilor, ci pe relația instrucțiune de executat-date de prelucrat. Un *flux* (stream) este definit ca o secvență de informații (instrucțiuni sau date) interpretate de un procesor. Rezultă patru clase, funcție de fluxul de instrucțiuni și date unice sau multiple :

(1) SISD — flux unic de instrucțiuni/flux unic de date. Acesta este calculatorul serial convențional, denumit și von Neumann, care execută un flux unic de instrucțiuni (pe o singură unitate procesoare). Fiecare instrucțiune aritmetică inițiază o operație aritmetică, conducînd la un flux unic de date cu argumente și rezultate legate logic. Este irelevant dacă se folosesc unități pipeline la execuția instrucțiunilor sau a operațiilor aritmetice. Acestea sînt calculatoare denumite anterior scalare. Iată cîteva exemple : CDC 6600 (fără pipeline), CDC 7600 (cu pipeline aritmetic), Amdahl 470 V/6 (pipeline pentru instrucțiuni).

(2) SIMD — flux unic de instrucțiuni/fluxuri multiple de date. Acesta este un calculator ce execută un flux de instrucțiuni, dar posedă instrucțiuni vectoriale care inițiază multe operații. Fiecare element al vectorului este interpretat ca aparținînd unui flux separat de date și de aici, exceptînd cazul degenerat al vectorilor cu un singur element, mai multe fluxuri de date. Această clasificare cuprinde deci toate mașinile cu instrucțiuni vectoriale. Din nou, este irelevant dacă procesarea vectorială este realizată pipeline sau cu masive de procesoare. Exemple : CRAY-1 (calculator vectorial pipeline) ; ILLIAC-IV (masiv de procesoare), ICL DAP (masiv de procesoare), OMEN 64 (masiv de procesoare).

3)) MISD — fluxuri multiple de instrucțiuni/flux unic de date. Această clasă pare a fi vidă, deoarece implică faptul ca execută mai multe instrucțiuni asupra aceluiași operand simultan. Flynn (1972) a menționat că include în această clasă structuri specializate ce folosesc mai multe fluxuri de instrucțiuni executate pe același flux de date. El nu dă exemple.

(4) MIMD — fluxuri multiple de instrucțiuni/fluxuri multiple de date. Fluxurile multiple de instrucțiuni implică existența mai multor unități de prelucrare a instrucțiunilor și în mod necesar mai multe fluxuri de date. Această clasă include toate formele de configurații multiprocesor, de la rețelele de calcul de uz general, la masivele de procesoare.

Din punctul nostru de vedere, schema de clasificare prezentată suferă datorită generalității ei : toate calculatoarele paralele, exceptînd multi-

procesoarele, intră în clasa SIMD și nu face nici o distincție între calculatoarele pipeline și masivele de procesoare, care au totuși arhitecturi în întregime diferite. Aceasta se datorește clasificării după funcțiile generale (indiferent dacă există sau nu instrucțiuni vectoriale) și nu după arhitectură. În această carte noi sîntem, ca și arhitectul, interesați de studierea detaliilor de organizare și deci avem nevoie de o clasificare mai fină.

1.2.3. Taxonomia lui Shore

Spre deosebire de Flynn, Shore (1973) și-a bazat clasificarea pe modul cum este organizat calculatorul din părțile componente. Au fost identificate 6 tipuri diferite de mașini, fiecăreia atribuindu-se o cifră romană (vezi figura 1.3).

Mașina I Arhitectură convențională, von Neumann, cu o singură unitate de comandă (CU), o unitate procesoare (PU), memorie pentru instrucțiuni (IM) și memorie pentru date (DM). O citire a DM produce toți biții unui cuvînt, care sînt prelucrați în paralel de PU. PU poate conține mai multe unități funcționale care pot fi sau nu pipeline; această clasă include atît calculatoare scalare pipeline (de exemplu, CDC 7600) cît și calculatoarele pipeline vectoriale (CRAY-1), a căror similaritate a arhitecturilor a fost deja menționată.

Mașina II Este similară mașinii I doar că un ciclu de citire a DM produce un bit slice; PU este organizată pentru execuția operațiilor în mod serial pe bit. Dacă considerăm memoria ca un masiv bidimensional cu un cuvînt memorat pe linie, mașina II citește un slice vertical de biți în timp ce mașina I citește slice-uri orizontale. Exemple: ICL DAP și STARAN.

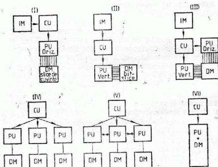


Fig. 1.3. Reprezentarea schematică a celor 6 clase de mașini de Shore (1973), Mașina I serială, pe cuvînt, paralelă pe bit; II, paralelă pe cuvînt, serială pe bit; III (= I + II), calculator ortogonal; IV, masiv neconectat; V, masiv conectat; VI, masiv cu logica în memorie.

Maşina III Este o combinaţie a maşinilor I şi II. Cuprinde o memorie bidimensională din care pot fi citite atît cuvinte cît şi bit slice-uri, un PU orizontal pentru prelucrarea cuvintelor şi un PU vertical pentru prelucrarea de bit slice ; pe scurt acesta este calculatorul ortogonal a lui Shooman (1970). Atît ICL DAP cît şi STARAN pot fi programate să funcţioneze ca maşini III, dar deoarece nu au PU separate pentru cuvinte şi bit slice, nu pot fi clasificate aici. Seria OMEN-60 este o implementare a maşinii III exact după definiţie (Higbie 1972).

Maşina IV Aceasta se obţine prin multiplicarea PU şi DM a maşinii I (definită ca un element procesor, PE) şi trimiţînd instrucţiunile acestui ansamblu de PE de la o singură unitate de control. Un exemplu bine cunoscut este PEPE. Absenţa legăturilor între PE limitează aplicabilitatea maşinii, dar face posibilă adăugarea de noi PE.

Maşina V Aceasta este maşina IV cu facilitatea suplimentară că PE sînt aranjate pe o linie şi se asigură conexiuni între vecinii cei mai apropiaţi. Fiecare PE poate adresa informaţii din memoria sa, dar şi în cea a vecinilor imediaţi. Exemplu este ILLIAC IV care asigură conexiuni între fiecare opt PE.

Maşina VI Maşinile I—V menţin conceptul separării între memoria de date şi unităţile procesoare, cu unele magistrale de date sau elemente de conectare între ele, deşi unele implementări ale maşinii II pe un bit (ICL DAP) includ PU şi DM pe aceeaşi placă. Maşina VI, ce implementează funcţii logice în memorie (logic — în — memory array sau LIMA) este o abordare alternativă a distribuirii comenzii în memorie. Exemple sînt de la memoriile asociative simple la procesoarele asociative complexe.

Se poate observa că maşinile II—IV sînt subdiviziuni folositoare ale clasei SIMD a lui Flynn, iar maşina I corespunde clasei SISD. Din nou, calculatoarele vectoriale pipeline, care necesită clar o clasă separată, nu sînt reprezentate satisfăcător, deoarece le-am găsit în aceeaşi clasă cu calculatoarele scalare ne-pipeline care nu au altă facilitate de paralelism decît execuţia operaţiilor aritmetice în paralel pe bit. De asemenea, nu satisface numerotarea claselor şi neutilizarea unor mnemonice semnificative.

1.2.4. O notaţie structurală

Primul pas în crearea unei scheme de clasificare este definirea unei notaţii adecvate. Ea trebuie să fie mai detaliată decît cele ale lui Flynn şi Shore pentru a diferenţia calculatoarele descrise şi a considera explicit arhitecturile pipeline. Notaţia este structurală şi se bazează pe o codificare ce indică numărul instrucţiunilor, unităţilor de execuţie şi memoriilor, modul lor de interconectare şi comandă. În acest sens se apropie de o formulă chimică, dar descrierea unui calculator este mai complexă : de exemplu, în chimie există un singur tip de atom de carbon, de notat că cei 3 izotopi naturali ai carbonului au aceeaşi structură electronică şi prin urmare aceeaşi chimie, în timp ce la un calculator trebuie distins între mai multe tipuri de unităţi de execuţie (pentru numere întregi, în virgulă mobilă, pipeline, bit serial etc.). O definiţie matematică a sintaxei notaţiei este formulată în Apendix 1, cu ajutorul formalismului Backus Naun (BNF).

Notația consideră un calculator format dintr-un număr de unități funcționale care manipulează date, conectate și operînd sub controlul unității centrale. Cea mai simplă notație a unui calculator serial von Neumann în notație structurală este:

$$C = I[E-M]$$

care definește calculatorul C ca fiind o unitate de prelucrare a instrucțiunilor I ce comandă unitățile din paranteză. Acestea sînt o unitate aritmetică E legată cu o cale pentru date (linia de legătură sau strangularea von Neumann) la o singură unitate de memorie M.

Se observă că notația noastră este similară cu PMS (Processor-Memory Switch) introdusă de Bell și Newell (1971), de care diferă prin gradul mai mare de adecvare la obiectivul urmărit. Obiectivul nostru principal a fost de a permite o descriere cu o singură expresie pe o linie, sau pe foarte puține linii, a arhitecturilor de ansamblu într-o manieră algebrică, similară programelor de calculator.

În particular dorim o notație îngrijită pentru masivele de procesoare de dimensiuni și tipuri diferite unde ne interesează în gradul cel mai înalt numărul și tipul unităților de manipulare a datelor și cum sînt comandate.

Notația structurală va folosi următoarele categorii:

A. *Unitățile* (regulile 1—10)—definesc simbolurile pentru reprezentarea unităților și combinarea lor în grupuri.

B. *Conexiunile între unități* (regulile 11—16)—definesc notația căilor pentru date și maniera de descriere a rețelelor complexe.

C. *Comentarii* (regula 17) — permite includerea de informații suplimentare într-un mod flexibil și deschis.

D. *Controlul unităților* (regulile 18—20)—definesc diferitele tipuri de control care fac dintr-un ansamblu de unități un calculator.

E. *Exemple* — prezintă cît de diferit pot fi descrise calculatoarele.

A *Unitățile*

(1) *Simbolurile* — se prezintă o listă alfabetică de simboluri pentru unitățile ce formează un calculator sau procesor:

B O unitate de execuție logică sau aritmetică în virgulă fixă sau mobilă.

C Un calculator care este o combinație de unități ce includ cel puțin o unitate I.

Ch Un canal de I/E care trimite date la sau primește date de la o unitate de interfață cu un dispozitiv de I/E și memorie, independent de alte unități.

D Un dispozitiv de I/E, de exemplu lectorul de cartele, unitatea de disc. Tipul dispozitivului este specificat de un comentariu în paranteză.

E O unitate de execuție care tratează date. Ea realizează funcții aritmetice logice și de manipulare a biților. Se împarte în unități de tip F și B, iar de obicei se numește ALU.

F O unitate de execuție în virgulă mobilă.

H O cale de date sau switch. Transferă datele fără a le modifica, deci fără posibila lor reordonare (ex., rețeaua FLIP de la STARAN).

I O unitate de interpretare a instrucţiunilor care decodifică un flux unic de instrucţiuni şi trimite (sau emite) comenzi unităţilor de execuţie. Controlează secvenţa instrucţiunilor cu un singur numărător de instrucţiuni. Este denumit adeseori IPU sau unitate de prelucrare a instrucţiunilor.

IO O interfaţă de dispozitiv de I/E care primeşte date de la un dispozitiv şi le înregistrează într-un registru local sau vice versa.

M O unitate de memorie uni-dimensională, de exemplu registre, memorii tampon, memoria principală, disc.

O O memorie bi-dimensională sau ortogonală.

P Un procesor definit la un ansamblu de unităţi, inclusiv o unitate E, dar fără unităţi I.

S Un switch care interconectează alte unităţi, ca de exemplu o reţea omega. De obicei nu execută operaţii asupra datelor.

U O unitate nespecificată. Este un simbol folosit cînd nu se poate aplica nici una din notaţiile anterioare. Natura unităţii este specificată de un comentariu în paranteză. Exemple sînt unităţile de control intermediar într-un sistem complex.

(2) *Pipelining* este indicat de litera mică „p" ce urmează simbolului unităţii, sau o structură în paranteză. Aceasta înseamnă că cel puţin o parte din operaţii sau suboperaţii se suprapun în timp, de exemplu :

Ip o unitate de prelucrare a instrucţiunilor pipeline;

Ep o unitate de execuţie pipeline;

{E < — —, — — > M} o structură în care citirea memoriei, execuţia operaţiilor aritmetice şi scrierea în memorie sînt suprapuse.

(3) *Instrucţiuni vectoriale* sînt indicate cu litera „v" după simbolul I, dacă apar în setul de instrucţiuni. În cazul masivelor procesoare, la care toate instrucţiunile sînt operaţii cu vectori, simbolul este subînţeles şi de aceea omis. De exemplu :

Ipv o unitate de prelucrare a instrucţiunilor pipeline ce execută instrucţiuni vectoriale

In o unitate ce execută instrucţiuni vectoriale, fără pipeline.

(4) *Unităţi diferite* de acelaşi tip pot fi separate cu un întreg la sfîrşit, de exemplu :

Ep1 unitatea de execuţie pipeline numărul 1

Ep3 unitatea de execuţie pipeline numărul 3;

(5) *Substituţia* este încurajată, iar pentru mai multă claritate se foloseşte notaţia ierarhică. De exemplu, tipurile definite cu regula (4) pot fi dezvoltate, cu utilizarea separatorului „;" :

I[E1, E2]; E1 = ; E3 = .

(6) *Unităţi multiple* — numărul unităţilor de acelaşi fel ce lucrează simultan se indică cu un număr întreg. În particular, o unitate complexă, deşi poate executa mai multe operaţii, dar la un moment dat execută una singură este considerată ca fiind una, de exemplu :

E unitate de execuţie multifuncţională pentru înmulţire, adunare, operaţii logice etc. ce realizează operaţiile la un moment dat (ca la IBM 7090);

10E 10 unităţi funcţionale (independente pentru înmulţire, adunare, operaţii logice etc. ce lucrează simultan ca la CDC 6600.

(7) *Multiplicarea*—o bară peste simbol, sau peste o structură delimitată de paranteze, { } indică că toate unitățile din grup sînt identice e.g.

$64\overline{p} = 64\overline{(E-M)}$ 64 elemente procesoare identice, ca la ILLIAC IV.

(8) *Grupuri de unități* sînt definite prin includerea unităților în paranteză. Dacă unitățile sînt separate de virgule (separator concurent) pot lucra simultan și deci în paralel. Dacă unitățile sînt separate de un „/" (separator secevențial) unitățile lucrează cîte una la un moment dat, adică secvențial, de exemplu :

{4Fp, 2B} înseamnă 4 unități aritmetice în virgulă mobilă pipeline și 2 unități nepipeline pentru operații cu întregi care lucrează simultan ;

{E1/E2/E3} trei unități de execuție care lucrează secvențial. Într-un set de paranteze se poate folosi un singur separator, deși o unitate multiplă implică operarea simultană, de ex. :

{3F1/B1} 3 unități în virgulă mobilă care lucrează simultan, dar secvențial cu o unitate în virgulă mobilă ;

3F1 = {F(+), F(*), F(+)} trei unități în virgulă mobilă ce lucrează concurent ;

B1 = {B(+)/B(shift)} unitatea în virgulă mobilă conține un sumator și un registru cu deplasare care lucrează secvențial. Conform regulii (17), simbolurile din paranteză rotundă sînt comentarii.

(9) *Numărul biților* prelucrați în paralel se indică cu un indice,

I_{16} o unitate ce execută instrucțiuni pe 16 biți.

E_{64} o unitate de execuție pe 64 biți.

Fluxurile multiple de instrucțiuni prelucrate de o singură unitate I, ca la Denelcor HEP, pot fi indicate de un multiplicant cu numărul posibil de fluxuri, de exemplu :

Ip_{50*64} unitate de prelucrare a instrucțiunilor în interiorul PEM de la Denelcor HEP, care prelucrează 50 fluxuri de instrucțiuni a 64 biți în manieră pipeline.

Dacă se omite multiplicantul, se consideră un singur flux. Notația ce urmează este folosită pentru unitățile de memorie. Se tratează fiecare secțiune a memoriei care lucrează separat ca o unitate independentă. Se folosește un asterisc pentru înmulțire, de exemplu :

nM_{w*b} o memorie unidimensională împărțită în n blocuri ; fiecare bloc conține w cuvinte de b biți, iar accesul se face pe b biți în paralel ;

M_{1k*32}o memorie de 1024 cuvinte a 32 biți ;

$8M_{64*64}$ cele 8 registre vectoriale de la CRAY-1, fiecare cu 64 cuvinte a 64 biți ;

$O_{\vec{w}*\vec{b}}$ o memorie bidimensională ortogonală de w cuvinte a b biți, iar accesul se face fie pe un cuvînt de b biți, fie pe un bit slice de w biți.

(10) Timpul caracteristic asociat cu lucrul unității se indică cu un exponent. Unitatea de măsură este nanosec., dacă nu se definește alta în text, de ex. :

I^{40} Unitatea de interpretare instrucțiuni cu un tact de 40 ns

E^{200} unitate de execuție cu un timp de operare mediu de 200 ns

M^{650} unitate de memorie cu un timp de acces de 650 ns

B. *Conexiunile între unități*

(11) *Magistralele de date* se indică cu următorii conectori :

— conexiuni de un tip nespecificat ;

< — conexiune simplă la stînga ;

— > conexiune simplă la dreapta;
< — > conexiune full duplex;
⟨—/—⟩ conexiune half duplex.

O conexiune simplă poate transfera datele numai în direcţia prezentată. O conexiune full duplex transferă datele în ambele sensuri simultan. O conexiune half duplex transferă datele în ambele sensuri, dar nu în acelaşi timp. Folosind separatorii concurent şi secvenţial, obţinem:

< — > o abreviere pentru {< —, — >}
< —/— > o abreviere pentru {<—/— >}

Linioara de legătură poate fi multiplicată pentru a creşte impresia de descriere structurală:

E——>M; E———>M; E<———/———>M

Săgeţile pot fi tipărite cu simbolurile „mai mic ca" şi „mai mare ca". De notat folosirea „;" (ca la ALGOL 60) ca separator între expresiile structurale. Mărimea căii de date poate fi menţionată sub conector în forma „număr de biţi" +(opţional) „numărul biţilor de adresă". Astfel,

$\overset{100}{\underset{64+16}{\text{———}}}$ bus de 64 biţi pentru date şi 16 biţi de adresă la fiecare 100 ns.

$\overset{->}{8}$ conexiune simplă la dreapta pe 8 biţi.

Magistralele multiple de date se descriu cu semnul de multiplicare (asterix), ca în FORTRAN. Se folosesc acoladele pentru grupări algebrice, deoarece parantezele rotunde sînt rezervate pentru comentarii. De exemplu:

$\overset{\langle\text{————————}\rangle}{4*\{64+16\}}$ 4 magistrale identice full duplex, fiecare cu 94 biţi de date şi 16 biţi de adresă.

O cale complexă sau switch este identificată special şi explicat mai apoi, de exemplu:

$$E—H3—M;\ H3 = \{\{\underset{16}{—}>,\ <\underset{8}{—}\}\ <\underset{24}{—}$$

Partea din dreapta a unei astfel de definiţii trebuie să conţină numai căi de date.

(12) *Conexiune în serie* — un lanţ de unităţi legate prin magistrale de date ca E—M1—M2—M3, descrie un set de unităţi ce sînt conectate logic în serie ca la un circuit electric.

(13) *Conexiune paralelă* — în sens logic o conexiune paralelă într-un circuit electric poate fi descrisă cu un separator concurent (,) sau secvenţial (/) între unităţi (sau descrieri ale căilor paralele) şi incluzînd lista între paranteze { }. Se obţine o flexibilitate mai mare prin introducerea simbolului nici o conexiune (|). O unitate poate fi conectată în afara parantezelor astfel:

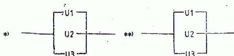
U conexiune externă nespecificată;
—U| conexiune externă la stînga, nici o conexiune la dreapta
|U— conexiune externă la dreapta, nici o conexiune la stînga;
—U— conexiuni externe la stînga şi dreapta.

Dacă nu există ambiguitate simbolul „|" poate fi omis; de exemplu dacă nu există nicăieri, conexiunea poate fi realizată în orice caz:

—{U1, U2, U3} un grup de unități care pot opera concurent, conectate într-un mod nespecificat la magistrala de date la stînga;

—{U1/U2/U3} — un grup de unități operînd secvențial cu conexiuni nespecificate la magistrala de date la dreapta și stînga;

—{—U1—, —U2—, —U3—} — trei căi paralele ce operează concurent și poate fi desenat: *)



—{U1—, |U2—, —U3—} — ca mai sus, dar U2 nu este conectat la stînga, astfel: **)

—{—U1—/—U2—/—U3—} — trei căi paralele alternative, lucrînd secvențial; deși acestea diferă ca operare în timp, nu diferă ca desen de al treilea exemplu de mai sus.

Pentru a ilustra utilizarea descrierii căilor acestor conexiuni paralele, considerăm următorul exemplu:
care poate fi descris cu {U7—, U8—, U9—, U10}—{{—U6—/<—}—{—U1—, |U2—, —U3—U4—, —U5|}/—>}—U11
unde săgețile sînt căi unidirecționale și alternative.

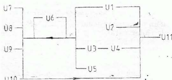
(14) *Punctele de conectare* sînt specificate cu litere mici în interiorul sau la sfîrșitul conexiunii; de exemplu:

$$E—a—M,\ U<—/—>q$$

Acestea pot specifica conexiuni lungi sau structuri care nu pot fi reprezentate în maniera ultimului paragraf. De exemplu, structura ce reprezintă trei calculatoare C1, C2, C3 conectate la 5 blocuri de memorie M1, M2, M3, M4, M5 este exprimată clar cu:

$$C = \{C1—\{a,\ b,\ c\},\ C2—\{a,\ c\},\ C3—\{b,\ c\},\ \{a—M1,\ a—M2,\ c—M3,\ b—M4,\ b—M5\}\}$$

unde punctele de conectare a, b, c au fost definite pe blocurile de memorie. În cazul conexiunilor multiple se observă utilizarea în paranteză a listei punctelor de conectare. Deci, punctele de conectare oferă posibilitatea descrierii unei rețele arbitrare de unități, în maniera prezentată.

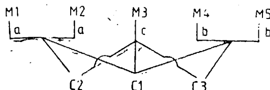


(15) *Masivele de procesoare* sînt de cele mai multe ori aranjate rectangular sau matriceal. Este deci natural să exprimăm un astfel de aranjament într-o formă multiplicativă ce precede des-

crierea procesorului. Folosim asterixul pentru multiplicare (ca la FORTRAN) și puteri; de exemplu:

128*64$\overline{P}$ un masiv de 8192 procesoare identice aranjate ca o matrice 128*64

$64^2\overline{P}$ un masiv de 64*64 procesoare



2^4 {2$\overline{C}$} un hipercub 4-dimensional cu 16 noduri. Fiecare nod are 2 calculatoare.

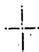
Extinderea conectivității între procesoare este reprezentată cu o mențiune la paranteze de formă „c—nn". Aceasta înseamnă că procesoarele pot transfera date direct la sau de la procesoare externe și includ vecinii cei mai apropiați de ordinul c. Dacă c=0 nu există conexiune directă. De exemplu:

288{3$\overline{E}$—$\overline{M}$}$^{0-nn}$ PEPE cu 288 procesoare neconectate, fiecare cu 3 unități de execuție și o memorie;

C[64*64$\overline{P}$]$^{1-nn}$ ICL DAP cu conexiuni la cei mai apropiați 4 vecini*);

{32*32$\overline{P}$}$^{2-nn}$ CLIP (Duff 1978) care are conexiuni la vecinii de ordinul 2. Aceasta înseamnă 8 vecini**).

Alte modele de conectare mai complexe pot fi specificate drept comentarii între paranteze.

*)  **) 

(16) *Conexiunea în cruce* (rețele cu accesuri încrucișate sau alte rețele de comutare) între unitățile de execuție și memoriile multiple se notează cu „×". Detaliile rețelei de conectare trebuiesc explicate într-un text separat. Timpii de transfer ca și numărul de biți transmis se notează ca mai înainte. De exemplu:

Ip[16$\overline{F}$ × 17$\overline{M}$] BSP cu 16 unități de execuție în virgulă mobilă conectate la 17 blocuri de memorie.

Simbolul „×" poate fi multiplicat pentru a crește lizibilitatea, iar astfel mai mulți „×" sînt identici din punct de vedere logic. Rolul exact îndeplinit de un switch poate fi definit prin explicitarea funcției sale într-un comentariu (vezi regula (17)). De exemplu:

$$nE \times \left(\left[\frac{x}{2^k}\right]2^k + \|x|_{2k} + 2^{k-1}|_{2k},\ k = 1, \ldots, \log_2 n\right) \times mM$$

reprezintă permutarea considerată în § 3.3.2 și prezentată în fig. 3.9. În expresia de mai sus $[f]$ este partea întreagă a lui f, iar $|x|_{2^k}$ înseamnă că argumentul x este luat modulo 2^k (vezi § 3.3). Se poate folosi ca alternativă și substituția pentru mai multă claritate, de ex.:

$$nE \times H3 \times mM;\ H3 = H(|x|2^{k+1})$$

Se poate specifica un switch şi prin folosirea simbolului S. Acest lucru este avantajos cînd switch-ul realizează conexiuni la o extremitate; de exemplu, cînd un număr de unităţi identice sînt conectate cu un switch într-un sistem MIMD sau cînd switch-ul este o unitate a sistemului. De exemplu,

C(Cedar cluster) = 8 $\overline{C}$×S (switch omega local)

C *Comentarii*

Notaţia permite un plus de claritate prin folosirea comentariilor.

(17) *Comentariile* sînt incluse în paranteze ( ) şi conţin informaţie opţională suplimentară despre simbolul precedent. Sintaxa este indiferentă, de ex.:

$M1^{10}$ (bipolar)—$M2^{400}$(MOS)—$M3^{(1ms)}$ (disc) o ierarhie de memorii;

Fp(*, ECL) un multiplicator în virgulă mobilă, pipeline şi tehnologie ECL;

Ip(4segs) [ ] un pipeline pentru interpretarea instrucţiunilor cu 4 segmente.

De notat utilizarea cometariului ca exponent pentru a specifica unitatea de timp. Simbolul nespecificat (U) permite definirea oricărui tip de unitate, de ex.:

U(2803) o unitate centrală IBM 2803.

În cazul conexiunilor pentru date parantezele intervin între linii sau „×", de ex.:

E—(half mile coaxial line)—M; E×(NBanyan network)×M.

Se folosesc simboluri, bit (b) şi bait (B) cu unităţile SI uzuale şi convenţia K=1024, M=K*K, G=K*K*K, T=K*K*K*K.

D *Comanda unităţilor*

Un număr de unităţi sub comanda unui flux de instrucţiuni definesc un calculator.

(18) *Calculatoare şi procesoare.* În cadrul notaţiei menţinem convenţia că un calculator este un grup de unităţi care pot prelucra instrucţiuni şi deci conţine cel puţin o unitate I. În contextul arhitecturii de ansamblu, o unitate I este programabilă şi prelucrează fluxul de instrucţiuni al utilizatorului. Cu această definiţie un microprocesor este descris mai bine ca un microcalculator. Cel mai simplu calculator este descris cu:

$$C=I[E-M]$$

Pe de altă parte, un procesor este orice mulţime de unităţi care pot prelucra date, dar nu prelucrează fluxul de instrucţiuni al utilizatorului. Deci, conţine o unitate E dar nu şi I. Cazul cel mai simplu ar fi:

$$P=E-M$$

Această definiţie a procesorului este în concordanţă cu utilizarea frecventă a expresiei masiv de procesoare pentru un masiv de unităţi E—M aflate sub comanda comună a unei unităţi I exterioare, ca la ICI DAP.

Distincţia de mai sus nu este rigidă, deoarece multe unităţi ,pe care noi le vedem ca unităţi de execuţie, sînt de fapt controlate prin microprograme, ale căror instrucţiuni sînt de fapt prelucrate de unităţile E. Din

punct de vedere al arhitecturii de ansamblu care ne interesează aici, problema importantă este dacă o unitate este programată de utilizator. Dacă da, vom considera că are o unitate I. Dacă nu, nu există deci unitate I deşi funcţionarea ei poate include un flux prestabilit de microinstrucţiuni. Similar, din acest punct de vedere, vom descrie registrele programabile ca părţi ale descrierii unui calculator, deşi pot exista şi alte registre interne ale calculatorului (de ex., între etajele unei unităţi de execuţie pipeline).

Nu este nici un motiv pentru care notaţia să nu fie folosită pentru descrierea în detaliu a structurii interne a unui pipeline aritmetic microprogramat sau microprocesor. În acest caz, toate registrele interne, programabile sau nu, vor fi descrise, iar unitatea de prelucrare a microprogramelor va fi clasificată ca o unitate I. Deci, este clar că înţelesul unei unităţi I depinde de utilizarea care i se dă notaţiei şi trebuie explicitată în text.

(19) *Extinderea comenzii* exercitată de unităţile I sau C este ilustrată de [ ]. Unităţile comandate sînt trecute în interiorul parantezelor, separate cu o virgulă dacă lucrează simultan sau cu „/" dacă lucrează secvenţial, de ex. :

I1[4I2[64]$\overline{P}$]] o unitate (I1) master comandă 4 blocuri, fiecare cu o unitate (I2) ce comandă 64 procesoare identice; proiectul original pentru ILLIAC IV;

Ip[C1, C2, C3] unitate I pipeline ce comandă 3 calculatoare care pot lucra simultan;

C1[Ep1/Ep2]; C1=I[B—M] un calculator C1 de comandă care efectuează operaţii booleene, conţine memorie şi comandă 2 unităţi de execuţie pipeline ce lucrează secvenţial.

(20) *Tipul comenzii* exercitată de unităţile I sau C poate fi indicat opţional cu indici, litere mici, ataşaţi la paranteze. Următoarele modalităţi sînt adecvate pentru descrierea calculatorului în acest volum. Altele pot fi descrise în comentarii :

asincron (a) unităţile comandate au mai mult de un generator de tact. Generatoarele nu sînt sincronizate, iar comunicaţiile între unităţi sînt coordonate de biţii de stare şi protocoale de transmisie.

orizontal (h) o instrucţiune compusă controlează funcţionarea unui set de unităţi diferite la fiecare tact, de ex. FPS AP-120B.

lockstep (l) o mulţime de procesoare identice comandate sincron pentru a realiza aceeaşi operaţie în acelaşi timp, de ex. ICL DAP.

issue-when-ready (r) instrucţiunile sînt transmise unităţilor de execuţie imediat ce acestea sînt libere, de ex. CRAY-1.

De exemplu :

I[10F, 10C]$_r$ CDC 6600 cu 10 unităţi funcţionale în virgulă mobilă diferite şi 10 calculatoare de I/O identice. Instrucţiunile sînt executate cînd unităţile sînt ready.

C[64$\overline{P}$]$_l$ Illiac IV cu 64 procesoare identice comandate sincron (lockstep mode).

I[4C]$_{\text{descrierea controlului}}$. Patru calculatoare comandate de o unitate I în manieră descrisă de comentariul din paranteză.

E. *Exemple*

Prezentăm mai jos ecuaţiile care descriu diverse calculatoare pentru a ilustra notaţia de mai înainte. Detaliile de descriere pot varia funcţie de necesităţi, de multe ori fiind de preferat o descriere ierarhică-arhitectura de ansamblu se prezintă în prima ecuaţie, iar detaliile apar în ecuaţii suplimentare, cînd este necesar pînă la nivelul registrelor individuale. Structurile mai complexe pot solicita extinderea formulei într-o a doua dimensiune, în maniera unei formule chimice.

$$C(Z80+\text{memory})=C(Z80)_{\bar{8}}M_{64*8}\ ;\ C(Z80)=I_8^{250}[B_{\overline{88}}M_{18*8}]$$

$$C(\text{INTEL } 8086)=I_8[B_{16\overline{16}}M_{13*8}]_{15}M_{1M*16}$$

$$C(\text{EDSAC1})=I[B_1-M]\ ;M=M_{512*35}$$

$$C(\text{IBM } 7090)=I[F_{38}-M]$$

$$C(\text{CDC } 6600)=I[10E\ 32M\ 1\overline{0P}]_r\ ;\ 10E=\{4F_{60},\ 6B\}$$

$$C(\text{CDC } 7600)=I[9Ep-M-16P]_r\ ;\ 9Ep=\{3Fp,6B\};$$

$$M=32MI-8M2$$

$$C(\text{IBM360/195})=Ip^{54}[2C-32M1^{160}(1KB)-16M2^{600}];$$

$$2C=\{C1.C2\};C1=I[3Fp-M];\ C2=I[B-M]$$

$$C(\text{CRAY-1})=Iv^{12}[12Ep^{12}-16M^{50}]\ ;_r\ 12Ep=\{3Fp_{64},\ 9B\}$$

$$C(\text{CYBER } 205)=Iv^{20}[4Fp_{64}-512M^{80}_{16K*32}]$$

$$C(\text{TIASC}(21PU,\ 4\ \text{pipe}))=2Ivp[2\overline{F}p]--8M-8\overline{P}$$

$$C(\text{ILLIAC IV}(4\ \text{quadrant}))=C1[4C2^{60}[64P]_1^{1-nn}];$$

$$P=F_{64}^{600}-M_{2K*64}^{400}$$

$$C(\text{PEPE})=C1[31[288\overline{\{3E-M\}}_1^{0-nn}];\ \ C1=C(\text{CDC7600})$$

$$C(\text{BSP})=I0p[16\overline{F}\times 17\overline{M}]_1$$

$$C(64\times 64\ \text{ICL DAP})=C[64^2P]_1^{1-nn}\ ;P_1\infty\ B_1-M_{4K*1}$$

$$C(\text{STARAN})=I[32[256\overline{B}_1\times O_{256*256}]_1]-M$$

$$C(\text{OMEN-64})=I_{16}[64\overline{B_1}-O_{64*16}-E_{16}]_1$$

$$C(\text{HYPERCUBE})=I[2^4\overline{C}];\ C=C(2*\text{INTEL } 8080)$$

$$C(\text{HEP})=16\ \ \overline{Cp}^{100}_{50*64}(\text{PEM})\times(\text{packet switch})\times 128\ \ \overline{M}^{100}_{1M*64}(\text{DMM})$$

$$C(\text{EGPA})=C(\text{control})[2^2\overline{C}(\text{boundary})[2^2\overline{C}(\text{array})]]$$

$$C(/\text{CAP})=C(\text{IBM host})[10\{-(\text{channel})-C(\text{FPS } 164/\text{MAX})\}]$$

$$C(\text{Cedar})=16C1(\text{cluster})\times S1(\text{global omega switch})\times 256\ \ M_{16M*6}$$

1.2.5 O clasificare structurală

Vom formula acum, cu ajutorul notației introduse, o clasificare atît a calculatoarelor seriale cît și paralele. Subdiviziunile acestei clasificări sînt prezentate în fig. 1.4. Acestea sînt, în continuare, dezvoltate pentru

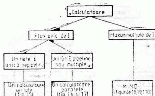


Fig. 1.4. O clasificare a arhitecturii de calculatoare în sens general.

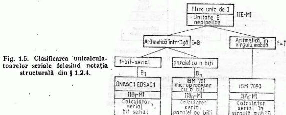


Fig. 1.5. Clasificarea unicalculatoarelor seriale folosind notația structurală din § 1.2.4.

calculatoarele seriale în fig. 1.5, iar pentru cele paralele în fig. 1.6 pînă la 1.10. Aceste figuri au o structură arborescentă și astfel, există o singură cale din vîrful diagramei la oricare din clasele de calculatoare definite pe ultima linie. Un calculator ce aparține la o clasă trebuie să posede practic toate proprietățile enumerate în casetele traversate pentru a atinge acea definiție. Fiecare clasă este ilustrată cu un calculator reprezentativ, împreună cu definiția canonică a clasei în notație structurală plus un nume descriptiv pentru calculatoarele acelei clase. Inevitabil anumite calculatoare au proprietăți ce aparțin la mai mult de o clasă. În această situație trebuie decis care este proprietatea dominantă. Sperăm că această situație se întîlnește rar, iar clasificarea este suficient de fină pentru a diferenția între calculatoarele care trebuie tratate separat.

La nivelul cel mai înalt vom respecta clasificarea funcțională a lui Flynn, în sensul împărțirii calculatoarelor în cele cu un singur flux de instrucțiuni (SI) și cele cu mai multe fluxuri de instrucțiuni (MIMD). În secțiunea §1.2.6 se prezintă o taxonomie pentru calculatoarele MIMD. Aici vom continua cu împărțirea mașinilor SI în cele cu o singură unitate I în, cele cu o singură unitate E ne-pipeline și în cele cu unități E multiple

şi/sau pipeline. Reamintim că o unitate poate executa numai o funcţie la un moment dat (chiar dacă poate executa mai multe funcţii) o unitate E ne-pipeline conduce (fig. 1.5) la modul de execuţie secvenţial, deci la toate calculatoarele seriale, în timp ce unităţile E multiple sau pipeline

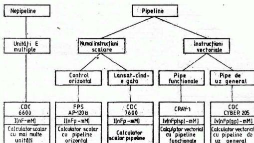


Fig. 1.6. Unicalculatoare paralele baazate pe paralelism funcţional şi pipelining. O clasificare a calculatoarelor prezentate în aceeastă carte.

permit diferite moduri de suprapunere a operaţiilor şi, în consecinţă familia unicalculatoarelor paralele (fig. 1.6. şi 1.7).

Următorul nivel al clasificării se bazează pe tipul de aritmetică realizat de unitatea E. Diferenţa de complexitate între o unitate aritmetică pe 1 bit (cu mai puţin de 10 porţi logice) şi o unitate aritmetică în virgulă mobilă (mai multe mii de porţi logice) este suficient de mare pentru a determina categorii calitative distincte ce trebuie recunoscute. Spaţiul suplimentar necesar unităţilor în virgulă mobilă ridică probleme privind asamblarea masivelor de procesoare diferite de cele ale sistemelor de procesoare pe 1 bit. În consecinţă, masivele de procesoare în virgulă mobilă şi masivele de procesoare pe 1 bit diferă ca implementare şi proprietăţi. Pentru a include evoluţia istorică a calculatoarelor seriale, această clasă a fost împărţită în sistemele cu aritmetică pentru numere întregi şi a celor cu aritmetică în virgulă mobilă, iar prima clasă în cea a calculatoarelor seriale şi a calculatoarelor paralele. Această parte a clasificării este mai mult decît de interes istoric, deoarece cuprinde şi primele generaţii de microprocesoare (de ex. microprocesoare de 4 biţi).

Fig. 1.6 ilustrează introducerea paralelismului funcţional şi pipelining în cadrul calculatoarelor seriale, de unde vom separa în primul rînd clasele de calculatoare cu şi fără unitate E pipeline. La stînga se află calculatoarele scalare cu unităţi multiple ne-pipeline, ca CDC 6600, care realizează în întregime performanţele sale prin paralelism funcţional. La dreapta calculatoarele pipeline se împart în cele cu sau fără instrucţiuni vectoriale explicite. Considerăm că această distincţie este necesară pentru a separa calculatoarele scalare de mare performanţă ca CDC 7600 de cele vecto-

riale pipeline precum CRAY-1. Din alte puncte de vedere aceste mașini sînt foarte asemănătoare. Calculatoarele pipeline cu instrucțiuni vectoriale se subîmpart în cele cu pipeline separate, specializate pentru fiecare tip de operație aritmetică (de ex. CRAY-1), și cele cu una sau mai multe uni-

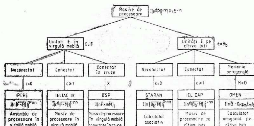


Fig. 1.7. O clasificare a masivelor de procesoare tratate în această carte (STARAN este clasificat pentru situația cînd rețeaua FLIP nu realizează permutarea datelor).

tăți pipeline de uz general, fiecare îndeplinind mai multe tipuri de operații (de ex. CYBER 205). Calculatoarele pipeline care au numai instrucțiuni scalare se împart în cele la care o instrucțiune controlează toate unitățile în fiecare ciclu (control orizontal ca la FPS AP-120B) și în cele în care instrucțiunile sînt lansate unităților în mod individual cînd sînt gata pentru a executa o operație (ca la CDC 7600).

Alternativa obținerii paralelismului prin replicarea procesoarelor sub control sincron-lockstep se prezintă în fig. 1.7. Din nou, acestea se împart în clasa calculatoarelor în virgulă mobilă și în clasa celor pe bit (inclusiv masivele cu control sincron care folosesc microprocesoarele de 8 biți). Sistemele de calcul se mai pot clasifica și după modul de conectare între procesoare, dacă sînt neconectate (conectivitate $c=0$) sau conectate la vecini ($c \geqslant 1$) într-un masiv d-dimensional. Alte forme de conectare aparțin clasei procesoarelor și memoriilor cu conexiuni încrucișate. În categoria procesoarelor pe bit se pot identifica sistemele asociative clasice cu procesoare neconectate, în timp ce sistemele care folosesc memorii ortogonale sînt menționate explicit. Multe calculatoare pot fi programate să adreseze atit „WORD SLICES" cît și „BIT SLICE", dar observați că pentru a respecta definiția noastră calculatorul ortogonal trebuie să aibă procesoare word slices și bit slices separate. Cu alte cuvinte, trebuie să fie implementat ca sistem ortogonal pentru a fi clasificat ca atare.

1.2.6. O taxonomie pentru calculatoarele MIMD

Numărul mare de propuneri diferite pentru calculatoare cu fluxuri multiple de instrucțiuni, făcute în ultimii ani, din care numai cîteva au fost descrise în §1.1.8, generează o confuzie privind proiectele de noi cal-

culatoare. Pentru a restabili ordinea, prezentăm în fig. 1.8, 1.9 şi 1.10 o taxonomie posibilă pentru aceste calculatoare MIMD (Hockney 1985 b, d). În cadrul ei am inclus numai calculatoarele controlate de fluxuri multiple de instrucţiuni convenţionale — aşa numitele calculatoare *control-flow*. În §3.2.2 sînt luate în considerare formele mai noi de control pentru exploatarea paralelismului, calculatoarele numite *data-flow* şi *reducţioniste*. Fig. 1.8 prezintă clasificarea generală în sisteme *pipeline, cu comutaţie* (switched) şi *reţea* (network). Fluxurile multiple de instrucţiuni pot fi prelucrate fie de o unitate pipeline sofisticată ce lucrează în time-sharing, fie de hardware separat (şi în mod necesar mult mai simplu) pentru fiecare flux. Prima alternativă este descrisă ca *MIMD pipeline* şi poate fi întîlnită la Denelcor HEP în configuraţie cu un singur PEM (vezi §3.4.4). Sistemele MIMD ce folosesc a doua alternativă se împart în mod natural în cele cu un switch separat şi identificabil (*switched MIMD*) şi în cele în care elementele de calcul sînt conectate într-o reţea recognoscibilă şi adesea expandabilă (*MIMD networks*).

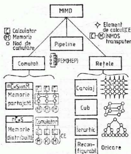


Fig. 1.8. O taxonomie structurală a calculatoarelor MIMD.

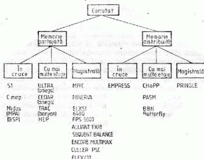


Fig. 1.9. Diviziuni ale clasei calculatoarelor MIMD cu comutaţie. Sub numele clasei se dau exemple de calculatoare. Cele din paranteze pătrate prezintă structura clasei, dar au control de tip SIMD, mai degrabă decît MIMD. În paranteză se specifică tipul comutatorului cu mai multe etaje.

La primele, toate conexiunile între calculatoare se realizează via comutator, care de obicei este destul de complex și formează o parte importantă a sistemului. La ultimele, elementele de calcul (CE) pot comunica direct numai cu vecinii din rețea ,iar comunicațiile la distanță mai mare solicită parcurgerea de către datele transmise a unui număr mare de CE intermediare. De aceea, CE trebuie să conțină un calculator mic (un microprocesor, de exemplu), o porțiune a memoriei sistemului și un număr de conexiuni la elementele vecine din rețea. Transputerul INMOS (vezi §3.5.5) avînd CE pe un cip, este un bloc de construcție ideal pentru *rețelele MIMD*.

Fig. 1.10. Diviziuni în cadrul clasei de calculatoare MIMD cu rețea.

În sistemele cu *rețea*, CE sînt noduri ale rețelei, de aceea mai pot fi denumite calculatoare nodale sau procesoare nodale sau elemente de procesare nodale. În cadrul notației și clasificării noastre, vor fi denumite elemente de calcul pentru a indica că sînt calculatoare complete, cu o unitate de prelucrare a instrucțiunilor. Termenul PE (processing element) este rezervat pentru combinația între o unitate aritmetică și memorie, fără unitate de prelucrare a instrucțiunilor, așa cum se întîlnește la calculatoarele SIMD ca ICL DAP (vezi §4.3.2).

În fig. 1.9 sistemele cu comutare sînt împărțite la rîndul lor în cele cu memorie distribuită tuturor calculatoarelor ca memorie locală, iar comunicația se execută via comutator (*distributed-memory MIMD*); în cele la care memoria este o resursă partajată, accesată de toate calculatoarele prin comutator (*shared-memory MIMD*). În continuare, mai este posibilă o împărțire funcție de natura comutatorului, iar în fig. 1.9 sînt prezentate ca exemple conexiunile cu *accesuri încrucișate*, conexiunile cu *mai multe etaje* și *magistrală*, atît pentru sistemele de tip shared-memory, cît și distributed-memory. Multe calculatoare mai posedă atît memorie partajată cît și distribuită local. Astfel de sisteme pot fi considerate ca hibrizi sau switched MIMD. Noi preferăm clasificarea lor ca variante ale sistemelor cu memorie partajată, deoarece rezervăm secțiunea cu memorie distribuită pentru sistemele fără memorie partajată. Clasificarea folosește localizarea memoriei ce păstrează permanent datele principale ale unei probleme. Memoria locală (sau cache) prezentă la aproape toate sistemele și folosită

pentru memorarea temporară în cursul calculatoarelor, nu este relevantă pentru clasificare.

Toate *rețelele MIMD* par să fie sisteme cu memorie distribuită, dar ele pot fi împărțite în continuare funcție de topologia rețelei, ca în fig. 1.10. Rețeaua cea mai simplă este în *stea*, în care mai multe calculatoare sînt legate la un calculator gazdă comun, ca la sistemul ICAP de la IBM Kingston și Roma. *Structuri simple sau multidimensionale* se pot întîlni la inelul Cyber Plus 1D, la NASA FEM și Columbia VEPP. Calculatoarele ICL DAP, Goddyear MPP și ILLIAC IV sînt exemple cu structuri 2D, dar ele sînt controlate de un singur flux de instrucțiuni și deci sînt mai mult calculatoare SIMD decît MIMD. *Rețelele hipercub* binare la care după fiecare dimensiune se întîlnesc numai 2 calculatoare formează o clasă interesantă ce a primit multă atenție în cazul proiectelor Cosmic Cube și a derivatului comercial, Intel iPSC. Mai există exemple de *rețele ierarhice* ce folosesc arbori, piramide (EGPA), ciorchini conectați la magistrale (Cm*). Rețeaua cea mai indicată va depinde cu siguranță de natura problemei de rezolvat, de aceea ideea unei rețele MIMD *reconfigurabilă* prin program este atractivă. Acest deziderat este considerat în cadrul calculatorului CHiP, ca și la ESPRIT Supernode de la Southampton (vezi § 3.5.5 (v)).

1.3 Analiza performanțelor

În secțiunea anterioară am discutat clasificarea arhitecturilor calculatoarelor de la microprocesorul scalar simplu, calculatoarele vectoriale pipeline, la masivele de procesoare. Aceste sisteme par a avea puține trăsături comune, ceea ce ne îndeamnă să definim doi parametri, în această secțiune, care caracterizează performanțele tuturor calculatoarelor seriale, pipeline și a masivelor de procesoare, considerate ca membrii diferiți ai unui spectru continuu mai mult decît ca sisteme fundamental diferite. Desigur, o astfel de analiză simplistă poate fi privită ca o descriere generală, dar credem că e suficient să întreprindem o analiză cantitativă a performanțelor calculatoarelor pentru a alege apoi algoritmul cel mai bun de aplicat. Pentru a extrage descrierea generală simplă a tuturor calculatoarelor seriale și paralele vom explica mai întîi, în detaliu, principalele căi de creștere a vitezei de calcul a unităților aritmetice.

1.3.1. Arhitecturi seriale, pipeline și masive de procesoare

Fig. 1.11 ilustrează modurile diferite de realizare a unei operații aritmetice de către arhitecturile seriale, pipeline și masive. Ca exemplu vom considera problema adunării a doi vectori în virgulă mobilă, x_i și y_i $(i = 1,2,\ldots,n)$ pentru a obține suma vectorială $z_i = x_i + y_i$ $(i = 1, 2,\ldots,n)$. Operația de adunare a oricărei perechi de elemente $(x = e2^p$ și $y_i = f2^q)$ este împărțită în patru suboperații care, pentru simplitate, vom considera că se execută în același interval de timp. Acestea sînt : (i) compararea exponenților, prin diferența $(p-q)$; (2) deplasarea lui x față de y cu $(p-q)$ poziții ; (3) adunarea mantiselor celor două numere ; și (4)

normalizarea rezultatului prin deplasarea lui z la stînga. La calculatoarele seriale cele 4 suboperaţii trebuie executate cu prima pereche x_1, y_1 pentru a produce primul rezultat z_1, înainte de a se prelucra următoarea pereche.

Modul de lucru secvenţial este ilustrat în centrul figurii 1.11, unde se presupune existenţa unei axe a timpului cu originea în partea de sus a figurii. Dacă l este numărul de suboperaţii (în acest caz l = 4) şi τ este timpul necesar pentru finalizarea fiecăreia (de obicei coincide cu perioada

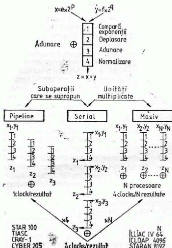


Fig. 1.11. Compararea arhitecturilor seriale, pipeline şi masive.

orologiului) atunci timpul necesar pentru a produce un vector de lungime n este

$$t_{serial} = l\tau n \tag{1.1a}$$

iar viteza maximă de obţinere a rezultatelor este

$$r_{\infty serial} = (l\tau)^{-1} \tag{1.1b}$$

În cazul prelucrării seriale, circuitele responsabile pentru fiecare din cele l suboperaţii sînt active numai i/l din timpul total. Această situaţie este

în sine ineficientă, mult mai evidentă dacă realizăm o analogie cu linia de asamblare a automobilelor. Suboperațiile necesare pentru obținerea sumei a două numere pot fi comparate cu suboperațiile necesare producerii unei mașini : de exemplu, (1) fixarea caroseriei pe șasiu ;(2) atașarea motorului ; (3) atașarea roților și (4) montarea ușilor. Prelucrarea secvențială corespunde unui singur grup de muncitori ce realizează o suboperație la un moment dat și unei singure mașini aflată în curs de asamblare. În mod clar, în exemplul nostru, trei sferturi din ansamblul liniei de asamblare-muncitori este nefolosit.
Linia de asamblare devine eficientă prin acceptarea unei noi mașini în suboperația (1) imediat ce mașina anterioară a trecut la suboperația (2). În acest mod este începută o mașină la fiecare τ unități de timp și, cînd linia este complet ocupată, se termină o mașină de fiecare τ unități de timp. De multe ori spunem că operațiile formează un pipeline, iar în exemplul nostru sînt 4 mașini în stagii diferite de asamblare pe linie și nici un muncitor nu stă degeaba. Acest principiu este folosit pentru creșterea vitezei de obținere a rezultatelor în unitățile aritmetice pipeline și este prezentat în stînga fig. 1.11. Diagrama de timp arată că creșterea de viteză se obține prin suprapunerea (prin execuția în același timp sau în paralel) a diferitelor suboperații cu diferite perechi de operanzi. Timpul necesar pentru obținerea, ca rezultat al unei operații, a unui vector de lungime n este prin urmare

$$t_{pipe} = [s+l+(n-1)]\tau \qquad (1.2a)$$

unde $s\tau$ este timpul de inițializare (set-up) necesar pentru încărcarea liniei, de a calcula prima și ultima adresă pentru fiecare vector ca și alte operații suplimentare. Este inclus, de asemenea, și timpul necesar transferului[1] numerelor între memorie și pipeline. l este numărul fazelor sau al segmentelor din pipeline și deci diferă pentru operații aritmetice diferite. Cînd este încărcat și deci operează continuu, un pipeline livrează un rezultat la fiecare t perioade de ceas, de aici

$$r_{\infty pipe} = \tau^{-1} \qquad (1.2b)$$

Comparînd acest rezultat cu ecuația (1.1b) se observă că se obține, în acest caz, o creștere a vitezei de cel mult l ori numărul suboperațiilor ce se execută simultan.

Din cele de mai sus este evident că orice operație care poate fi împărțită în suboperații de durată egală, poate fi implementată în pipeline. Un exemplu obișnuit este unitatea pipeline de execuție a instrucțiunilor la care se suprapun următoarele faze : (1) decodificarea instrucțiunii ; (2) calculul adresei operanzilor ; (3) extragerea operanzilor ; (4) trimiterea comenzilor unităților funcționale ; și (5) extragerea următoarei instrucțiuni. Exemple : **IMB 360/91, ICL 2980, AMDAHL 4C0V/6.** Alte calculatoare, ca **BSP**, suprapun operații ca (1) extragerea din memorie ; (2) operații aritmetice ne-pipeline și (3) memorarea rezultatelor. Mașinile care au un pipeline aritmetic nu au în mod necesar un repertoriu de instrucțiuni vectoriale (de ex. **CDC 7600, IBM 360/195**). Mașinile cele mai importante cu unitate pipeline aritmetică și instrucțiuni vectoriale sînt **CDC STAR 100, CYBER 205, TIASC** și **CRAY-1.**

O altă posibilitate de a crește viteza unității aritmetice este de a multiplica unitățile de execuție și de a forma un masiv de elemente procesoare (PE) sub controlul unui singur flux de instrucțiuni. Toate PE execută aceeași operație aritmetică în același timp, dar asupra unor operanzi, diferiți, stocați în memoriile lor locale. Dacă sînt N astfel de procesoare și $N<n$, primele N perechi (x_i, y_i) pot fi prelucrate simultan, într-un interval de timp pe care îl notăm cu $t_{||}$ (4τ în ex. nostru). Apoi sînt încărcate următoarele N elemente pentru a produce N rezultate în $t_{||}$ unități de timp, ș.a.m.d. Diagrama de timp este prezentată în partea dreaptă a fig. 1.11. Concludem că timpul necesar pentru a calcula un vector de lungime n cu un astfel de calculator este

$$t_{array} = t_{||} \lceil n/N \rceil \quad (1.3a)$$

și

$$r_{\infty array} = N/t_{||} \quad (1.3b)$$

unde $\lceil x \rceil$ este cel mai mic întreg mai mare sau egal cu x. Funcția dă numărul de repetiții necesare dacă vectorul are mai multe elemente decît numărul de procesoare. Viteza maximă de calcul se atinge cînd n este un întreg multiplu de numărul de procesoare; se obține o creștere a vitezei de N ori față de un procesor serial cu același tip de unitate aritmetică.

Calculatoarele au evoluat sub forma procesoarelor în virgulă mobilă nepipeline, ca **ILLIAC IV** (**64PE**), **BSP** (**16PE**), cît și ca masive foarte mari de procesoare pe 1 bit : STARAN (256PE) și ICL DAP (4096PE). Ultima alternativă devine de mare interes cînd se implementează în tehnologie VLSI, ideală pentru multiplicarea pe scară largă a structurilor logice simple. Există și posibilitatea unor variante de mijloc constituite din masive de microprocesoare comerciale existente (de 4, 8 și 16 biți). De asemenea, pentru atingerea unor anumite performanțe, se poate alege între cîteva unități pipeline aritmetice optimizate și un masiv de procesoare mai simple. Diferențele sînt evidente pentru următoarele sisteme comerciale competitive : **CRAY-1** (2 unități pipeline aritmetice), BSP (un masiv de 16 procesoare în virgulă mobilă ne-pipeline) și ICL DAP (un masiv de 4096 procesoare pe 1 bit). Principiile de calcul pipeline și de multiplicare a procesoarelor pot fi îmbinate într-o o structură de masiv de microprocesoare pipeline. în jurul anilor 85, deși erau în uz sisteme cu unități pipeline multiple (de ex. **CDC CYBER 205** și **CRAY X-MP**), nu existau masive multidimensionale de procesoare pipeline.

Lucrarea (Graham 1970) realizează o prezentare bună a acestor două concepte. O analiză mai detaliată a arhitecturii pipeline, incluzînd discutarea unor proiecte actuale, poate fi găsită în Ramamoorthy și Li (1977) și Kuck (1978), iar aspectele inginerești, în Jump și Ahuja (1978) ca și în lucrarea lui Koogge (1981), denumită *„The Architecture of Pipelined Computers"*. Arhitecturile de masive sînt analizate de Thurber și Wald (1975) și de Thurber (1976).

1.3.2. Parametrii de performanță (r_∞, $n_{1/2}$)

Vom folosi drept criteriu unic pentru arhitecturile diferite ce au fost descrise în secțiunea anterioară, performanțele calculatoarelor pentru o singură operație aritmetică cu un vector de lungime n. Aceasta va fi expri-

mată cît mai apropiat de formula generală a timpului de operare, t, ca o funcţie de lungimea vectorului :

$$t = r_{\infty}^{-1}(n + n_{1/2}) \qquad (1.4a)$$

Cei doi parametri, $n_{1/2}$ şi r_{∞} descriu complet performanţele hardware ale unui calculator generic ideal şi dau o primă descriere a oricărui calculator real. Aceşti parametri caracteristici se numesc :

(a) *lungimea corespunzătoare performanţei jumătate* (the half-performance length $n_{1/2}$)-dimensiunea vectorului necesar pentru a atinge jumătate din performanţa maximă ;

(b) *performanţa maximă sau asimptotică* r_{∞}—viteza maximă de calcul în unităţi de operaţii în virgulă mobilă efectuate pe secundă. Pentru un calculator ideal acest parametru are o creştere asimptotică odată cu creşterea numărului de componente ale vectorului (lungime) spre infinit. Unitatea de măsură pentru execuţia în virgulă mobilă este milionul de operaţii în virgulă mobilă pe secundă (megaflop/s sau Mflop/s).

Dacă $n < n_{1/2}$ este mai convenabil să se folosească formula echivalentă

$$t = \pi_0^{-1}[1 + (n/n_{1/2})] \qquad (1.4b)$$

unde $\pi_0 = r_{\infty}/n_{1/2}$ este denumită performanţă specifică.

Cînd se deduc valorile medii pentru $n_{1/2}$ şi r_{∞} corespunzătoare unei secvenţe de operaţii vectoriale trebuie să ne amintim că t este definit ca timpul necesar pentru *operarea unui vector de lungime n.* Astfel, dacă a operaţii aritmetice vectoriale consumă timpul

$$T = b + cn \qquad (1.4c)$$

atunci

$$t = T/a = (c/a)\ (n + b/c)$$

De aici, prin comparaţie cu ecuaţia (1.4a)

$$r_{\infty} = a/c,\ n_{1/2} = b/c \qquad (1.4d)$$

de unde este clar că sporind viteza tuturor circuitelor calculatorului cu acelaşi factor *f* (prin împărţirea lui b şi c cu *f*), de exemplu prin micşorarea perioadei ceasului, se creşte performanţa asimptotică cu acelaşi factor *fără* a modifica $n_{1/2}$.

Semnificaţia celor doi parametri este destul de diferită. (r_{∞}) caracterizează în primul rînd tehnologia folosită. Este un factor de scală aplicat performanţelor unui calculator particular care reflectă tehnologia în care a fost implementat. Mai mult, dacă comparăm performanţele (definite ca proporţionale cu inversul timpului de execuţie) diferiţilor algoritmi executaţi pe acelaşi calculator, r_{∞} nu joacă aici un rol în alegerea algoritmului cel mai bun. De aceea acest parametru nu intervine în discuţia algoritmilor din cap. 5.

Pe de altă parte, lungimea performanţei jumătate ($n_{1/2}$), este o măsură a paralelismului ce caracterizează arhitectura calculatorului. Vom vedea că variază între $n_{1/2} = 0$ pentru calculatoare seriale şi fără nici o operaţie paralelă şi $n_{1/2} = \infty$ pentru un masiv infinit de procesoare. Astfel, se realizează o apreciere cantitativă a paralelismului arhitecturii calculatoarelor. Deoarece $n_{1/2}$ nu apare ca factor în ecuaţia (1.4a), compararea algoritmilor diferiţi executaţi de un calculator se întreprinde prin aprecierea valorii lui $n_{1/2}$. Lungimea vectorului (sau lungimea medie), n, măsoară paralelismul problemei, iar raportul $v = n_{1/2}/n$ arată cît de paralel este un calculator în raport cu o anumită problemă. Dacă $v = 0$ sau mai mic, atunci un algoritm proiectat pentru un mediu secvenţial sau serial va fi cel mai indicat, iar dacă v este mare algoritmul proiectat pentru un mediu paralel se dovedeşte a fi cel mai bun .Cap. 5 cuprinde o analiză a influenţei lui $n_{1/2}$ sau v asupra performanţelor algoritmilor.

Este evident, pe baza ecuaţiei (1.2a), că pentru un calculator pipeline orice overhead, ca de exemplu timpul de iniţializare, contribuie la valoarea lui $n_{1/2}$, deşi nu reprezintă o caracteristică arhitecturală. Numărul etajelor în pipeline, l, din această expresie, este o măsură a paralelismului hardware, deoarece reprezintă numărul operaţiilor executate în paralel. Deci, nu este adevărat în mod absolut că $n_{1/2}$ măsoară întotdeauna paralelismul hardware, dar îl putem defini ca o măsură a *paralelismului aparent* al hard-ului. Din punctul de vedere al utilizatorului, comportarea calculatorului este determinată de expresia (1.4a). Un calculator pipeline cu un $n_{1/2}$ mare se comportă ca şi cînd ar avea un grad înalt de paralelism hardware, chiar dacă aceasta s-ar datora unui timp de iniţializare mai lung.

Şi, pentru utilizator, nici nu contează cît de mult din paralelismul aparent este real. Din acest motiv nu vom face în continuare o distincţie între paralelismul real şi aparent; ne vom referi la $n_{1/2}$ ca la o măsură a paralelismului calculatorului.

Formulind ecuaţia (1.4a) în termenii timpului de iniţializare, t_0, şi a timpului necesar pentru obţinerea unui rezultat, τ, obţinem :

$$t = t_0 + nt = \tau(n + t_0/\tau) \tag{1.4e}$$

care, prin comparaţie cu ecuaţia (1.4a), corespunde la :

$$n_{1/2} = t_0/\tau = t_0\tau_\infty \tag{1.4f}$$

unde

$$\tau_\infty = \tau^{-1}$$

Comparaţia ecuaţiilor (1.4a) şi (1.4e) conduce la alte 2 interpretări ale semnificaţiei lui $n_{1/2}$. Prima, $n_{1/2}$ măsoară numărul de operaţii în virgulă mobilă care s-ar fi putut executa în cursul timpului de iniţializare t_0. Deci, măsoară importanţa, în termenii operaţiilor în virgulă mobilă pierdute, timpului de iniţializare pentru utilizator. A doua, cînd lungimea vectorului este egală cu $n_{1/2}$, primul şi al doilea termen al ecuaţiei (1.4e) sînt egali, şi jumătate din timp s-a consumat pentru iniţializare (primul termen), iar cealaltă jumătate pentru execuţia unor operaţii aritmetice folositoare (al doilea termen).

Analize anterioare ale timpului de execuţie efectuate de Calahan (1977), Calahan şi Ames (1979), Heller (1978) şi Kogge (1981) folosesc mai degrabă expresii liniare ca ecuaţia (1.4e), şi nu expresia (1.4a). Toţi aceşti autori recunosc importanţa raportului (t_0/τ), dar nu-l folosesc ca parametru de prim ordin. Noi considerăm că pentru compararea calculatoarelor şi a algoritmilor, timpul absolut de iniţializare nu este de importanţă majoră, ci raportul lui la r necesar pentru obţinerea unui rezultat. De aceea, folosim pentru acest raport simbolul $n_{1/2}$, pe care îl utilizăm ca element central al analizei noastre.

Din cele de mai sus este clar că o analiză a valorii $n_{1/2}$ poate fi aplicată oricărui proces care respectă o relaţie liniară ca (1.4a) sau (1.4e). Un caz evident este cel al operaţiilor de intrare/ieşire (I/E). În cazul multor probleme timpul necesar operaţiilor de I/E domină timpul de calcul. Obţinerea datelor de pe disc se face cu un timp mare de iniţializare necesar pentru deplasarea braţului şi căutarea pistelor. Cu alte cuvinte, timpul necesar accesării a n elemente respectă ecuaţia (1.4e), care este cel mai bine interpretată prin folosirea ecuaţiei echivalente (1.4a) şi a parametrilor r_∞ şi $n_{1/2}$. Viteza maximă, r_∞, se măsoară în MB/s iar $n_{1/2}$ ar fi lungimea blocului măsurat în octeţi, necesar pentru atingerea unei viteze de transfer medii 12 r_∞. De obicei, valorile lui $n_{1/2}$ sînt foarte mari pentru sistemele de I/E, ceea ce arată că sînt de dorit transferurile cît mai rare ale unor blocuri mari de date ($n > n_{1/2}$).

1.3.3. Măsurarea lui $n_{1/2}$ şi r_∞

Cei doi parametri pot fi analizaţi cel mai bine prin determinarea experimentală a timpilor de execuţie pentru probleme de test :

```
      CALL SECOND (T1)
      CALL SECOND (T2)
      T0=T2-T1
      DO 20 N=1,NMAX
      CALL SECOND (T1)
      DO 10 I=1,N
   10 A(I) = B(I)*C(I)
      CALL SECOND (T2)
   20 T=T2-T1-TO
```

sau codul în limbaj de asamblare echivalent. Bucla DO 10 de mai sus ar fi înlocuită de orice compilator vectorial cu o instrucţiune vectorială. Măsurarea şi scăderea timpului de apel al subrutinei, TO, este necesar pentru o măsurătoare de calitate. NMAX este lungimea maximă a vectorului, iar SECOND este o subrutină de calcul al timpului UC în secunde. Dacă se trasează apoi t funcţie de n se obţine o dreaptă ca în fig. 1.12. Intersecţia cu axa n, în domeniul valorilor negative, dă valoarea lui $n_{1/2}$, iar inversul pantei de pe dreaptă dă valoarea lui r_∞. Rezultatele prezentate în figură sînt obţinute cu un CDC CYBER 205 şi sînt tipice pentru o maşină pipeline.

Valoarea lui $n_{1/2}$ se poate afla și din informațiile furnizate de producători. De obicei acestea vor fi sub forma ecuației (1.6a).

$$t = \tau(s + l + (n - 1)) \qquad (1.6a)$$

de unde, prin comparație cu ecuația (1.4a) se obține pentru un calculator pipeline

$$n_{1/2} = s + l - 1 \qquad (1.6b)$$

și

$$r_\infty = \tau^{-1} \qquad (1.6c)$$

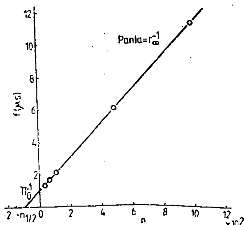


Fig. 1.12. Măsurări ale parametrilor n1/2 și r∞ pentru un calculator, funcție de lungimea vectorului. Datele sînt obținute din lucrarea Kascic (1979) pentru CYBER 203E cu două unități pipeline; $n1/2 = 100$, $r_\infty = 10$Mflop/s.

Deoarece lungimea unității pipeline, l, depinde de operația ce este executată, $n_{1/2}$ nu poate fi absolut constant pentru un anumit calculator. Va depinde de modul de realizare a operației și de modul de utilizare a calculatorului. În particular, codul necesar ce apare în controlul buclelor, introdus de compilator, va apare ca un adaos software la valoarea lui s și de aici o valoare crescută pentru $n_{1/2}$. Cu aceste onservații, considerăm $n_{1/2}$ un parametru util pentru caracterizarea performanțelor vectoriale.

Pentru calculatoarele seriale, comparînd ecuația (1.1a) cu formula generală (1.4a) obținem

$$n_{1/2} = 0 \text{ și } r_\infty = (l\tau)^{-1} \qquad (1.6d)$$

Caracterizarea masivelor de procesoare cu parametrul $n_{1/2}$ este mai puțin evidentă, deoarece formula (1.3a) este discontinuă, lucru ilustrat și de fig. 1.13. Cel mai bine este să se distingă două cazuri funcție de lungimea vectorului n în raport cu numărul procesoareor N. Dacă $n \leqslant N$ masivul este încărcat total sau parțial o singură dată. Astfel, timpul pentru o operație paralelă este independent de n și egal cu $t_{||}$. În această circumstanță, din punct de vedere al problemei de rezolvat, masivul apare ca și cum ar avea un număr infinit de procesoare. Limita corectă se obține cu formula generală (1.4b) dacă luăm

$$n_{1/2} = \infty \text{ și } \pi_0 = t_{||}^{-1} \text{ pentru } n \leqslant N \qquad (1.7a)$$

Pe de altă parte, dacă $n > N$, sistemul va fi încărcat de mai multe ori, iar cea mai bună caracterizare o obținem dacă luăm ca aproximare generală o linie ce reprezintă comportarea medie a sistemului. Aceasta este

linia întreruptă din fig. 1.13 care trece prin centrul pașilor ce definesc performanța actuală. Se obține:

$$n_{1/2} = N/2 \text{ și } r_\infty = N/t_{||} \quad \text{pentru } n > N \qquad (1.7b)$$

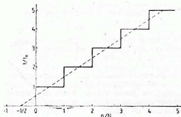


Fig. 1.13. Timpul t pentru operații cu vectori de lungime n pe un masiv de N procesoare (linia plină). Aproximația liniară cea mai bună (linia întreruptă) arată că $n_{1/2} = N/2$ și $r = N/t_{||}$, unde $t_{||}$ este timpul necesar unei operații paralele executate de masiv.

O situație mult mai complicată intervine în cazul masivelor de procesoare pipeline — de exemplu CDC NASF. În acest caz timpul de execuție este

$$t = \tau[(s + l - 1) + \lceil n/N \rceil] \qquad (1.8a)$$

unde N este numărul de pipeline aritmetice identice, iar $s\tau$ și l sînt timpul de inițializare și numărul de segmente ale fiecărui pipeline. Acest rezultat poate fi înțeles gîndindu-ne la sisteme ca niște pepilene ce execută suboperații cu supercuvinte, fiecare format din N numere. Atunci, $\lceil n/N \rceil$ este numărul acestor supercuvinte care trebuie prelucrate și înlocuiește pe n în ecuația (1.6a). Această ecuație este reprezentată grafic în fig. 1.14 (linia plină) împreună cu aproximarea generală (linia întreruptă), de unde concluzionăm:

$$n_{1/2} = N(s + l - 1/2) \text{ și } r_\infty = N/\tau \qquad (1.8b)$$

Fig. 1.14. Măsurarea lui $n_{1/2}$ pentru un masiv cu N unități pipeline. Linia plină reprezintă timpul actual, iar cea întreruptă pe cel generic. În acest exemplu, $n_{1/2} = 5{,}5N$, $r = N/t_{||}$ și $s + l = 6$.

Neglijînd diferența nesemnificativă între 1 și 1/2 din for-

mulele(1.6b) și (1.8b) pentru $n_{1/2}$, vedem că multiplicînd unitatea pipeline de N ori, se obține numai o majorare a ratei maxime de N ori, dar și a lungimii corespunzătoare performanței medii de același număr de ori. Ultimul rezultat era de așteptat dacă interpretăm $n_{1/2}$ ca expresie a cantității de paralelism în sistem, adică numărul total de operații ce se realizează în paralel. Paralelismul a N unități pipeline este de așteptat să fie atunci de N ori mai mult decît paralelismul unui singur pipeline.

1.3.4. Spectrul calculatoarelor

Cei doi parametri, $n_{1/2}$ și r_∞ ne permit să comparăm din punct de vedere cantitativ paralelismul și performanțele maxime ale tuturor calculatoarelor. Este foarte instructiv să analizăm poziția relativă a diferitelor calculatoare în planul ($n_{1/2}$, r_∞). O astfel de diagramă care poate fi privită ca definind un spectru al calculatoarelor, este reprezentată în fig. 1.15 și cuprinde calculatoarele prezentate în această carte. Axa verticală r_∞ este performanța maximă în megaflop/s, iar axa orizontală ($n_{1/2}$) exprimă paralelismul arhitecturii. Astfel, un calculator este reprezentat de un punct al planului. Deoarece, de cele mai multe ori, un calculator este utilizat în moduri diferite cu nivele variabile de paralelism, el poate fi reprezentat cu o linie continuă ce leagă tocmai aceste puncte. Cu linii întrerupte se reprezintă evoluțiile parcurse de diferitele arhitecturi. Utilizînd o scară logaritmică, se marchează în mod arbitrar sistemele seriale ($n_{1/2} = 0$) pe linia $n_{1/2} = 1$. Această alegere este acceptabilă deoarece, cînd prelucrează vectori, calculatoarele seriale prezintă o valoare a lui $n_{1/2}$ mică, diferită de 0.

Am menționat că multiplicînd un procesor de N ori, atît $n_{1/2}$ cît și r_∞ cresc de N ori și, în consecință, sistemele ce se obțin prin multiplicare se vor găsi pe aceeași dreaptă la un unghi de 45°. Ca exemplu, citām modurile de operare ale sistemului ICL DAP reprezenta în dreapta jos; versiunile cu 1, 2 și 4 unități pipeline ale sistemului **TIASC** și cu 2 și 4 unități pipeline ale lui **CYBER 205**, reprezentate în centrul figurii; și evoluția proiectului BSP cu 16 PE în proiectul **NASF Burroughs** cu 512 PE. De-a lungul acestor linii raportul $r_\infty/n_{1/2}$, performanța pe unitatea de paralelism, este constantă. Acest raport notat cu π_0, este mai specific unei anumite familii de calculatoare decît $n_{1/2}$ sau r_∞ considerate separat. Este de dorit ca acest raport să fie mare, deoarece este necesar mai puțin paralelism pentru atingerea unei performanțe dorite. Totuși, sînt necesare tehnologii mai avansate și metode de răcire corespunzătoare pentru a crește π_0 și, astfel, costurile cresc rapid. În tabelul 1.2 se prezintă π_0 pentru o gamă de calculatoare, în ordinea descrescătoare a valorilor. De asemenea, performanța specifică este parametrul care determină performanța unui calculator pentru vectori mici (vezi §1.3.5). De aici, performanțele pentru această situație nu se schimbă de-a lungul diagonalelor, și cresc numai la traversarea lor ortogonală, spre extremitatea superioară stînga. Progresele tehnologice, în particular reducerea perioadei ceasului, duc la creșterea lui r_∞ fără a modifica $n_{1/2}$ și produc deplasări verticale în plan. Acest proces este evident în evoluția de la **CYBER 205** (20 ns) la **CDC**

NASF (10 ns). O reducere a timpului de inițializare produce o reducere a lui $n^{1/2}$ și o deplasare spre stînga. Aceasta contribuie de asemenea la modificarea proiectului CDC NASF plecînd de la **CYBER 205**.

Fig. 1.15. Reprezentarea în planul ($n_{1/2}$, r_{∞}) a spectrului de calculatoare. Liniile diagonale la aproximativ 45 corespund la dezvoltarea prin multiplicare. Operațiile sînt în virgulă mobilă. Apostroful marchează numărul de biți folosit la reprezentarea unui număr.

Gama de calculatoare pe care o prezentăm variază de la ICL DAP ($n_{1/2}$ ridicat) cu un număr mare de unități aritmetice lente identice, la calculatoare cu cîteva unități pipeline aritmetice foarte performante cum

este de exemplu CRAY-1 ($n_{1/2}$ scăzut). Astfel, **128×128** DAP și **CRAY-1** pot atinge aceeași performanță maximă, dar lungimea performanței medii a acestor mașini diferă cu două ordine de mărime și mai ales au caracteristici de programare diferite — **CRAY-1** are aproape caracteristicile unei mașini seriale ($n_{1/2}$ scăzut), iar **DAP** cele ale unei mașini infinit paralele

Tabelul 1.2 **Performanța specifică π_0 pentru o serie de arhitecturi de calculatoare paralele. Se specifică valorile maxime**

Calculator	Performanța specifică maximă $\pi_0 = r_\infty/n_{1/2}$ (M/s)
CDC 7600, CRAY-1	10
BSP, Burroughs NASF	7
CYBER 205, CDC NASF	1–4
TIASC	0,4
STAS 100	0,2
ICL DAP	0,008–0,04
Multi-microprocesoare (anul 1980) de ex. AMD AM 9511	0,008–0,02

($n_{1/2}$ ridicat). Cele două mașini sînt separate astfel de un spațiu mare în figura prezentată. Axa orizontală acoperă gama întreagă a arhitecturilor de la cele seriale, la cele foarte paralele, la dreapta. Se menționează că un calculator rezolvă eficient numai acele probleme ce sînt caracterizate de o lungime a vectorului mai mare decît $n_{1/2}$ propriu. Astfel, cu cît va fi mai mare valoarea lui $n_{1/2}$ cu atît va fi mai limitat numărul problemelor rezolvate eficient. În acest sens, axa $n_{1/2}$ poate fi interpretată ca acoperind gama calculatoarelor de la cele cu utilizarea cea mai generală, la stînga, la cele mai specializate, la dreapta.

1.3.5. Performanța matricială (SIMD)

Pînă acum am considerat performanța maximă r_∞ în cazul ideal al vectorilor de lungime infinită. Formula generală (1.4a) poate fi folosită pentru a introduce alți parametri care definesc performanțele calculatoarelor pentru vectori cu lungime finită. Acestea sînt:

(a) *performanța medie vectorială*

$$r = n/t = r_\infty/(1 + x^{-1}) = r_\infty \text{pipe}(x) \tag{1.9a}$$

$$\text{sau } r = \pi_0 n/(1 + x)$$

unde $x = (n/n_{1/2})$; și $\text{pipe}(x) = (1 + x^{-1})^{-1}$ și

(b) *eficiența vectorială*

$$\eta = r/r_\infty = (1 + x^{-1})^{-1} = \text{pipe}(x) \tag{1.9b}$$

În fig. 1.16 se reprezintă relația dintre eficiența vectorială și lungimea vectorului. Se observă că $\eta = 0.5$ pentru $n = n_{1/2}$ și că eficiența crește asimptotic spre 1 cu creșterea lungimii vectorului spre infinit. Este important de remarcat că apropierea de asimptotă se face lent, de exemplu, pentru o lungime a vectorului de $n = 4n_{1/2}$ atinge 80% din performanța maximă, iar pentru $n = 9n_{1/2}$ se atinge 90%. Panta inițială a curbei eficienței care determină eficiența pentru vectori scurți ,se reprezintă cu linie întreruptă. Aceasta ar uni punctul ($n = n_{1/2}$, $\eta = 1$) cu originea. Expresia funcțională $y = (1 + x^{-1})^{-1}$ intervine frecvent în analizele de performanță pentru a descrie modul de apropiere de asimptotă (vezi §1.3.6). Datorită utilizării ei, în contextul descrierii performanței unei unități pipeline, o denumim *funcția pipeline*, pipe(x). Cînd intervine, performanța poate fi definită cu 2 parametri, valoarea asimptotică (aici r_∞) și parametrul performanței medii (aici $n_{1/2}$, dar în altă parte $f_{1/2}$(§1.3.6) sau $s_{1/2}$(§1.3.6). Pentru masivele de procesoare curba eficienței este discontinuă, reprezentată cu o linie continuă în fig. 1.17. Oricum, linia întreruptă realizează o aproximare bună cu $n_{1/2} = N/2$ pentru acest tip de calculator.

Să analizăm performanțele unui calculator pentru vectori mari și mici, în comparație cu lungimea performanței medii. Să considerăm următoarele două situații limită :

(i) *Cazul vectorului lung* ($n \gg n_{1/2}$ și $x \to \infty$).
Avem din ecuațiile (1.4a) și (1.9a)

$$t = n/r_\infty \quad r = r_\infty \tag{1.9c}$$

Astfel, timpul de execuție este proporțional cu lungimea vectorului, iar performanța este constantă. Ecuația (1.9c) este ecuația de timp pentru calculatoarele seriale, unde timpul se obține împărțind timpul total pentru efectuarea operațiilor aritmetice la viteza de calcul. Astfel, în acest caz, orice calculator se comportă ca unul serial chiar dacă posedă un paralelism substanțial și $n_{\frac{1}{2}}$ este mare.

(ii) *Cazul vectorlului scurt* ($n \ll n_{1/2}$ și $x \to \emptyset$)
folosind relația (1.4b) și a doua ecuație (1.9a),

$$t = \pi_0^{-1}; \; r = \pi_0 n$$

Astfel, timpul de execuție este constant, iar performanța este proporțională cu lungimea vectorului. Aceasta este comportarea unui masiv infinit de elemente procesoare, deoarece există, în orice situație, suficiente procesoare pentru a fi asignate elementelor vectorului. Întotdeauna calculul poate fi încheiat în cursul intervalului de timp necesar execuției unei operații paralele a masivului, independent de lungimea vectorului. Acum toate calculatoarele apar ca masive paralele infinite, chiar dacă $n_{1/2}$ poate fi destul de mic.

Deci, r_∞ caracterizează performanța unui calculator, pentru vectori lungi, în timp ce π_0 caracterizează performanța pentru vectori scurți, de unde indicele 0.

Multe calculatoare paralele posedă o unitate scalară cu $n_{1/2} = 0$ și au o viteză de execuție maximă r ca și un masiv de prelucrare a vectorilor

sau un pipeline cu $n_{1/2} > 0$, caracterizat de o viteză de prelucrare maximă r_∞. Dimensiunea vectorului limită, n_b este dimensiunea vectorului de la care procesorul vectorial execută operațiile vectoriale mai rapid decît cel scalar. Folosind formula generală (1.4a) se obține

$$n_b = n_{1/2}/(R\infty - 1) \tag{1.10}$$

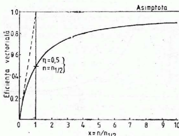


Fig. 1.16. Eficiența vectorială η ca funcție de lungimea vectorului n. Rata actuală de prelucrare este rata maximă × eficientă. Această curbă, $y=(1+x^{-1})^{-1}$ este denumită funcția pipeline, pipe(x).

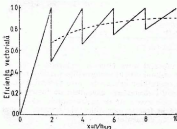


Fig. 1.17. Eficiența vectorială a unui proces de calcul pe un masiv de procesoare (linia continuă) în comparație cu reprezentarea ei aproximativă pentr-un calculator generic (linia întreruptă)

unde $R_\infty = (r_{\infty v}/r_{\infty s})$ este raportul între vitezele maxime vectorială și scalară, amîndouă măsurate în elemente pe secundă. Relația (1.10) este reprezentată în fig. (1.18). În general este de dorit să avem un n_b mic, astfel

vor fi puține probleme pentru care procesorul vectorial va fi folosit. Ecuația (1.10) arată că n_b mic este echivalent cu un raport mare între vitezele vectorială și scalară. n_b, ca și alți parametri, este folosit pentru selecția calculatoarelor ilustrată în tab. 1.3.

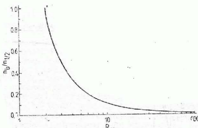


Fig. 1.18. Influența raportului R_∞ asupra lui n_b.

Tabelul 1.3. **Parametrii caracteristici pentru anumite calculatoare paralele**

Calculator *	$n_{1/2}$	r_∞ (Mflop/s	R_∞	n_b
64' CRAY-1	10—20	80	13	1,5—3
48' BSP	25—150	50	20	1—8
2-pipe 64'CDC CYBER 205	100	100	10	11
1-pipe 64' TIASC	30	12	4	7
64'C DC STAS 100	150	25	12	12
32' (64×64) ICL DAP	2048	16	400	5

* Prim notează biți

Deoarece raportul între vitezele vectorială și scalară este de obicei substanțial (de ordinul lui 10) performanța totală a unui program depinde de fracția v a operațiilor aritmetice între perechi de numere (pe care le vom numi *operații elementare*) executate de instrucțiunile vectoriale în comparație cu cele executate de instrucțiunile scalare. Ne vom referi la acest raport ca la fracția operațiilor vectorizate. Atunci timpul mediu pentru o operație elementară este:

$$t = vt_v + (1 - v)t_s \tag{1.11a}$$

unde t_v și t_s sînt timpii medii necesari pentru o operație elementară executată fie cu o instrucțiune vectorială, fie cu una scalară. Viteza de execuție este $r = t^{-1}$ și atinge un maxim r_v pentru vectorizarea completă ($v = 1$, $t = t_v$), iar fracția cîștigului maxim obținut cu o fracție v din operațiile vectorizate este:

$$g = r/r_v = (R + v(1 - R))^{-1} \tag{1.11b}$$

unde $R = r_v/r_s = t_s/t_v = R_\infty \eta$ este raportul curent între viteza vectorială și scalară pentru problema analizată, depinzînd în mod implicit prin η de lungimea vectorului. Fig. 1.19 prezintă g ca o funcție de v pentru R luînd valori între 2 și 1000. Este clar că pentru R mare aritmetica calculatorului trebuie vectorizată în proporție mare pentru a atinge performanța dorită. Obținerea unor nivele de vectorizare mari nu este așa dificilă cum apare, deoarece introducerea unei instrucțiuni vectoriale de lungime n vectorizează n operații elementare, unde n poate avea valori mari. Ca o măsură a cantității de vectorizare necesară, definim $v_{1/2}$ ca fracția de aritmetică ce trebuie vectorizată pentru a atinge jumătate din cîștigul maxim realizabil. Din ecuația (1.11b) obținem

$$v_{1/2} = (R-2)/(R-1) \tag{1.12a}$$

care pentru valori mari ale lui R devine

$$v_{1/2} = 1 - R^{-1} \simeq \text{pipe}\ (R) \tag{1.12b}$$

Această relație este desenată în fig. 1.20.

În cazul limită al unei unități vectoriale infinit de rapide ($R \to \infty$), găsim din ecuația (1.11b):

$$r = r_{\infty s}/(1-v) \tag{1.12c}$$

Dacă unitatea vectorială este mult mai rapidă ca cea scalară, viteza de execuție a calculelor vectoriale este determinată *numai* de viteza unității ei *scalare*, $r_{\infty s}$ și de către *fracția de operații scalare* din totalul de operații

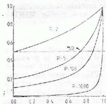


Fig.1.19. Influența fracției de aritmetică vectorială $v_{1/2}$ asupra fracției cîștigului maxim care se realizează g. R este raportul între vitezele vectorială și scalară, $v_{1/2}$ este valoarea lui v pentru un cîștig de jumătate.

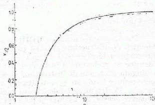


Fig. 1.20. Fracția de aritmetică ce trebuie vectorizată pentru a obține jumătate din cîștigul maxim $v_{1/2}$, ca funcție de R.

aritmetice $(1-v)$. Formulat altfel, dacă o broască țestoasă și un iepure sînt într-o cursă, viteza medie a perechii este determinată cu precădere de viteza broaștei și de distanța de parcurs. Viteza iepurelui nu este importantă deoarece timpul lui este, oricum, neglijabil. Situația nu se va îm-

bunătăţi semnificativ dacă iepurele va fi înlocuit de un ghepard. Acest efect este ilustrat de apropierea lentă a lui $v_{1/2}$ de valoarea asimptotică în fig. 1.20, cu creşterea vitezei unităţii vectoriale. Pe scurt, calculatoarele vectoriale cu unităţi scalare lente sînt condamnate, cum s-a văzut cu **CDC** STAR 100 (vezi §1.1.3). Efectul descris mai sus şi ecuaţiile (1.11) şi (1.12) sînt cunoscute în ansamblu ca *legea lui Amdahl*. IBM a tras concluzia, folosită la calculatoarele sale vectoriale, că nu este eficient să se construiască unităţi vectoriale de mai mult de 4—5 ori mai rapide decît unităţile scalare, deşi în tab. 1.2 s-a văzut că multe calculatoare vectoriale cu succes comercial au R_∞ semnificativ mai mare.

Legea lui Amdahl intervine cînd timpul total al unei activităţi este suma timpului unui proces rapid şi timpului unui proces lent. Deci, această lege se aplică cînd un job este distribuit pentru execuţie unui multiprocesor **MIMD**. Această problemă este deseori denumită organizarea *multitasking* a unui job. Dacă numărul procesoarelor este mare, să spunem N, atunci viteza de execuţie cu N procesoare este de N ori mai mare decît a acelor părţi ale job-ului executat secvenţial de un singur procesor. Astfel, pentru N mare, viteza medie de execuţie a unui job este determinată în primul rînd de viteza unui singur procesor (de obicei foarte lent dacă N este mare) şi de fracţia din job care nu poate fi paralelizată şi deci trebuie executată secvenţial, din nou ecuaţia (1.12c) (Larson 1984, 1985).

1.3.6. Performanţa sistemelor MIMD

Este important să înţelegem că adoptarea soluţiei programării mai multor fluxuri de instrucţiuni se face cu un anumit cost. Programarea MIMD creează overhead suplimentar cu efectele asociate şi uneori se pot chiar pierde avantajele cîştigate prin hardware. Dacă soluţia **MIMD** a unei probleme este sau nu avantajoasă depinde în mod efident de mărimea acestor costuri.

Dacă, într-un calculator **MIMD**, p microprocesoare execută o secvenţă identică de instrucţiuni (acelaşi parcurs prin acelaşi program), atunci p instrucţiuni identice sînt extrase din memorie şi furnizate la p unităţi identice de prelucrare a instrucţiunilor (IPU), cînd de fapt sînt necesare o singură extragere de instrucţiuni şi o singură **IPU**. În acest caz se risipesc resurse hardware şi se realizează un trafic cu memoria, care nu este necesar. Pentru un astfel de job este mult mai indicat un calculator SIMD, deoarece acesta ar extrage şi prelucra într-adevăr o singură instrucţiune în unitatea centrală master, în timp ce cele p elemente procesoare ar executa operaţii identice sub control central. Economiile la extragerea instrucţiunilor şi prelucrarea lor sînt cunoscute ca economii **SIMD** şi reprezintă motivul principal pentru care au fost inventate aceste calculatoare. Suprafaţa de siliciu economisită poate fi folosită pentru creşterea paralelismului unităţii aritmetice şi deci, pentru creşterea vitezei, în timp ce traficul redus la memorie ar reduce numărul conflictelor de acces la memorie şi întîrzierile asociate.

Este foarte probabil că sistemele **MIMD** vor executa programe identice cu toate microprocesoarele lor, căci cine ar scrie, de exemplu, o mie de subrutine diferite pentru un sistem MIMD cu 1000 procesoare

Desigur, punctul crucial este că nu se execută simultan instrucţiuni identice, chiar dacă programele sînt aceleaşi, datorită ramurilor de execuţie dependente de date ce vor determina parcursuri diferite prin programul executat de diferitele microprocesoare. Un prim exemplu este simularea Monte Carlo unde, prin alegerea aleatoare a datelor, se realizează în mod deliberat parcursuri diferite la fiecare execuţie a aceluiaşi program. Astfel de probleme impun în mod clar rezolvarea pe calculatoare MIMD, nefiind potrivite pentru sistemele SIMD.

Există, totuşi, multe situaţii cînd împărţirea naturală a unei probleme pe un calculator **MIMD** conduce la execuţia unei secvenţe identice de instrucţiuni de către toate microprocesoarele. Să considerăm, de exemplu, rezolvarea a p sisteme independente de ecuaţii tridiagonale, fiecare sistem fiind alocat unui procesor, sau calculul a p transformate Fourier independente. Ambele probleme intervin în faze diferite ale rezolvării ecuaţiilor diferenţiale parţiale cu transformate (vezi §5.6.2) şi nu conţin ramuri dependente de date. De aceea, sînt ideal de executat pe calculatoare SIMD, neavînd nevoie de un sistem **MIMD**.

Cele trei probleme (sau overheadul) asociate calculatoarelor MIMD sînt :

(1) *planificarea* activităţilor între procesoarele disponibile (sau fluxuri de instrucţiuni) într-un astfel de mod încît să se reducă, preferabil la 0, timpul cît procesoarele sînt inactive, aşteptînd alte procesoare ;

(2) *sincronizarea* procesoarelor astfel încît operaţiile aritmetice să se desfăşoare în secvenţă corectă ;

(3) *comunicarea* datelor între procesoare astfel ca operaţiile aritmetice să se realizeze cu datele corecte.

Problema comunicaţiilor între memorie şi unităţile aritmetice este prezentă la toate calculatoarele şi generează diferenţele între performanţele maxime ale unităţilor pipeline anunţate de producători şi performanţele medii realizate pentru probleme reale. Programarea şi sincronizarea sînt probleme noi ridicate de calculatoarele MIMD. Fenomenele enunţate sînt cuantificate cu 3 parametri, E_p pentru planificare, $s_{1/2}$ pentru sincronizare şi $f_{1/2}$ pentru comunicaţie (Hockney 1987b, c, d).

(i) *Parametrul planificării* : E_p

Planificarea este problema cea mai studiată referitor la calculatoarele MIMD ; într-adevăr, pînă de curînd, era singura problemă care a primit multă atenţie. Cele mai multe lucrări privitoare la algoritmii paraleli tratează problema planificării activităţilor diferiţilor algoritmi pe un sistem cu p procesoare, cu presupunerea că timpul necesar sincronizării şi comunicaţiilor poate fi ignorat. Pentru moment vom face aceeaşi presupunere, deoarece vrem să tratăm ultimele două efecte separat. Urmînd cercetările grupului condus de D. Kuck, de la Universitatea Illinois (Kuck 1978, p. 33), introducem *eficienţa planificării*, E_p activităţilor a p procesoare, aşa cum urmează. Dacă T_1 este timpul necesar unui singur procesor pentru execuţia tuturor activităţilor, iar T_p timpul pentru aceleaşi activităţi cînd sînt executate de p procesoare identice, avem :

$$E_p = T_1/(pT_p) \leqslant 1 \tag{1.13}$$

Se obține o planificare perfectă cînd este posibil să se aloce (1/p) din activități fiecărui procesor, cînd $T_p = T_1/p$ și $E_p = 1$. Dacă activitățile nu pot fi repartizate între cele p procesoare, atunci unele vor încheia operațiile înaintea altora, devenind inactive, și $T_p > T_1/p$ iar $E_p < 1$. Din această cauză planificarea mai este uneori referită ca *echilibrarea încărcării* (load balancing).

Dacă r_∞ este performanța maximă a unui sistem cu p procesoare, atunci fiecare procesor are o performanță maximă de r_∞/p. De asemenea, dacă procesoarele sînt calculatoare seriale ($n_{1/2} = 0$), de exemple microprocesoare, iar dacă activitățile constau în s operații aritmetice în virgulă mobilă, atunci intervalul de timp necesar unui procesor va fi:

$$T_1 = s/(r_\infty/p) = (sp)/r_\infty \qquad (1.14)$$

iar pentru p procesoare ce lucrează în paralel va fi:

$$T_p = T_1/(pE_p) = s/(r_\infty E_p) \qquad (1.15)$$

(ii) *Sincronizarea sau parametrul mărimii granularității* (grain size):

Introducem în formula (1.15) overhead-ul de sincronizare prin definirea conceptului de *segment de lucru* (work segment) ca unitate de bază din care sînt construite programele MIMD . Un segment de lucru este un set de instrucțiuni aflate între două puncte de sincronizare, programat sub forma a p task-uri complet independente, cîte unul pentru fiecare procesor. „Independent" înseamnă că nu există comunicație (transfer de date) între task-urile segmentului de lucru. Întinderea segmentului de lucru este determinată de faptul că se află între două puncte de sincronizare. Cu alte cuvinte, orice activitate inițiată înaintea segmentului de lucru trebuie să se încheie înaintea lansării acestuia în execuție, care, la rîndul lui, trebuie executat înainte de a se continua execuția programului. În programele științifice, dimensiunea activităților este de obicei egală cu numărul operațiilor în virgulă mobilă din segment.

Dacă dimensiunea activităților ce urmează să fie executate în paralel în mod independent de către p procesoare este mare, spunem că *granularitatea* paralelismului este mare, sau că programul este definit de paralelism cu granularitate mare. Din contra, dacă dimensiunea activităților împărțite este mică, spunem că paralelismul are granularitate mică. Diferența principală între programările SIMD și MIMD constă în tipul granularității paralelismului programului. La calculatoarele SIMD, sincronizarea se realizează prin hardware după fiecare operație vectorială, iar granularitatea paralelismului este egală cu lungimea vectorului, de obicei mic. La calculatoarele **MIMD**, dimensiunea granularității este de obicei mult mai mare și poate implica o întreagă fază a unui algoritm (de exemplu, transformata Fourier pentru o structură tridimensională). Putem cuantifica ideea de granularitate prin egalarea cu s, cantitatea de lucru într-un segment, așa cum a fost definit în paragraful anterior. Formulată astfel, dimensiunea granularității programelor SIMD constă în numărul instrucțiunilor buclelor DO cele mai interioare, care sînt înlocuite cu instrucțiuni vectoriale. Programele **MIMD** sînt împărțite de obicei la nivelul buclelor

DO cele mai exterioare, iar dimensiunea granularității va fi egală cu segmente mari din program.

Realizarea sincronizării impune un anumit interval de timp, t_0, ce depinde de mecanismele hardware, sistemul de operare și limbajul de programare. *Parametrul sincronizării*, $s_{1/2}$, se definește prin cantitatea de operații aritmetice care s-ar fi putut executa (la viteza maximă de r_∞) în timpul necesar sincronizării (Hockney 1985c). Astfel, $s_{1/2} = r_\infty t_0$ este, cu alte cuvinte, cantitatea de operații aritmetice pierdute datorită necesităților de sincronizare. Deci, acest parametru măsoară importanța overhead-ului de sincronizarea pentru utilizator, deoarece se măsoară în unități de operații aritmetice pierdute, în cursul execuției programului utilizatorului. Dacă adunăm timpul de sincronizare la timpul calculat anterior, T_p, se obține timpul necesar execuției unui segment de lucru

$$t_w = t_w t_0 + T_p = r_\infty^{-1}[(s/E_p) + s_{1/2}] \tag{1.16}$$

Pentru o problemă care poate fi planificată perfect ($E_p = 1$), ecuația (1.16) are pentru calculul MIMD aceeași expresie ca și ecuația (1.4a) pentru o instrucțiune vectorială SIMD, cu cantitatea de operații aritmetice s, analoagă lungimii vectorului n, iar $s_{1/2}$ este analog cu $n_{1/2} \cdot s_{1/2}$ este, de asemenea, cantitatea de operații aritmetice necesare într-un segment de lucru pentru ca viteza de calcul medie să fie jumătate din cea maximă, cu alte cuvinte $r_\infty/2$. În acest caz, cînd $s = s_{1/2}$, se consumă pentru operații aritmetice utile numai jumătate din timp, cealaltă jumătate fiind consumată pentru sincronizare. Deci, este folosit numai 50%din timp.De aceea, parametrul $s_{1/2}$poate fi denumit *dimensiunea granularității pentru jumătate din performanță* (half performance grain size). Performanța medie pentru alte valori ale cantității de operații aritmetice s se obține cu formula :

$$\bar{r} = \frac{s}{t_w} = \frac{r_\infty E_p}{1 + (s_{1/2}E_p/s)} = r_\infty E_p \mathrm{pipe}(s/s_{1/2}E_p) \tag{1.17}$$

Această expresie este identică cu ecuația (1.9a), reprezentată în fig. 1.16 pentru performanța vectorială medie ca funcție de lungimea vectorului, cu r_∞ înlocuit de $r_\infty E_p$, iar $n_{1/2}$ înlocuit de $s_{1/2}E_p$.

Este important să cunoaștem valoarea lui $s_{1/2}$ pentru un calculator **MIMD**, deoarece este o măsură ce poate fi folosită pentru judecarea dimensiunii granularității paralelismului programului în mod efectiv. Cu alte cuvinte, spune programatorului cît de multe operații aritmetice trebuie să fie într-un segment de lucru înainte de a împărți activitățile între mai multe procesoare. Dacă considerăm că o utilizare sub 50% este inacceptabilă, atunci prin definiție dimensiunea granularității (sau segmentului), s, trebuie să depășească $s_{1/2}$. Dacă, pe de altă parte, dorim atingerea lui s_b (break even grain size), valoare de la care este mai rapidă execuția job-ului cu p procesoare și acceptarea overhead-ului de sincronizare, decît fără sincronizare și utilizarea unui singur procesor, se obține prin egalarea ecuațiilor (1.14) și (1.16) :

$$s_b = s_{1/2}/(p - E^{-1}{}_p) \tag{1.18}$$

Pentru o planificare perfectă ($E_b = 1$), acest rezultat este perfect analog lungimii vectorului n calculat cu ecuația (1.10) și reprezentat în fig. 1.18, cu numărul procesoarelor p înlocuind R_∞.

Ca şi $n_{1/2}$, parametrul de sincronizare, $s_{1/2}$, trebuie să fie considerat ca o cantitate experimentală ce trebuie măsurată pe orice sistem **MIMD**, iar astfel de măsurători s-au efectuat pentru **CRAY X—MP** cu 2 CPU, **IBM /CAP** şi sînt prezentate în capitolul 2 (Hockney 1985a, 1987d), iar pentru **Denelcor HEP** în capitolul 3 (Hockney şi Snelling 1984, Hockney 1985c). Programele utilizate diferă ca detalii, datorită necesităţilor de sincronizare şi multi-tasking ale fiecărui sistem. Principiul, însă, este următorul: ca şi la măsurarea lui $n_{1/2}$ în programul (1.5), se alege ca task o înmulţire vectorială element cu element, ce urmează a fi împărţită în fluxuri de instrucţiuni. Cînd lungimea vectorului este un multiplu de numărul fluxurilor de instrucţiuni (sau procesoare), execuţia poate fi planificată perfect, şi în acest caz măsurăm timpul t ca o funcţie de cantitatea de operaţii aritmetice s:

$$t = r_\infty^{-1}(s + s_{1/2}) \qquad (1.19a)$$

iar performanţa medie este dată de funcţia pipeline

$$r = r_\infty \text{ pipe } (s/s_{1/2}) \qquad (1.19b)$$

Aceeaşi subrutină este furnizată fiecărui flux de instrucţiuni, constînd din bucla DO 10 a programului (1.5) cu parametrii specificînd valoarea de start şi sfîrşit pentru I. Acestea sînt stabilite de programul de control master pentru fiecare flux de instrucţiuni, pentru a se asigura că fiecare flux realizează o secţiune diferită din operaţiile de înmulţire vectoriale iniţiale.

Se pot distinge cazuri diferite pentru măsurarea sincronizării corespunzătoare diferitelor primitive de sincronizare asigurate, iar într-un sistem cu reţea, funcţie de faptul că datele necesare sînt în procesor sau trebuie transferate prin reţea. În ultimul caz, în măsurători se include şi costul comunicării. Pentru a se studia numai sincronizarea, ar fi necesar să se presupună că datele se află în memoria locală. Un studiu pentru valorile limită ale lui r_∞^{-1} şi $s_{1/2}$ furnizează un număr de valori între care s-ar putea situa valorile aplicabile problemelor actuale. La un sistem strîns conectat, unde se execută comunicaţii rapide între procesoare, domeniul cuprins între cazurile cel mai bun şi cel mai rău ar trebui să fie destul de mic; dar într-un sistem slab conectat, unde comunicaţiile între procesoare sînt lente, domeniul de valori poate fi mare. Variaţia lui $s_{1/2}$ cu întîrzierile datorate comunicaţiilor se tratează cu testul de eficienţă în domeniul $(\hat{r}_\infty, \hat{s}_{1/2}, f_{1/2})$, descris în §1.3.6, partea (iv).

(iii) *Parametrul comunicaţiei*: $f_{1/2}$

În cursul tuturor testelor folosite mai sus pentru măsurarea lui $n_{1/2}$ şi $s_{1/2}$ am considerat ca problemă standard înmulţirea vectorială element cu element şi am distins cazuri diferite pentru valori $(r_\infty, n_{1/2}, s_{1/2})$, depinzînd de faptul că vectorii erau memoraţi în registre locale sau memoria operativă. Dacă viteza de calcul a unei unităţi pipeline aritmetice este mai mare decît viteza de transfer a datelor cu memoria operativă, va interveni o strangulare a accesului la memorie (sau a comunicaţiei), iar viteza de calcul va creşte odată cu creşterea numărului de operaţii aritmetice per acces la memorie. Această situaţie pe care o vom desemna ca bandă a accesului

la memorie este caracteristică supercalculatoarelor, deoarece ele posedă unități aritmetice pipeline de mare performanță. În cazul testului nostru, la un acces la memorie se execută numai 1/3 dintr-o operație în virgulă mobilă, cazul cel mai defavorabil. Multe calcule permit o valoare mult mai mare, ca în analizele ce urmează.

Accesele la memorie pot fi modelate aproximativ (Hockney 1985d, Curington şi Hockney 1986) prin presupunerea că fiecare operație vectorială la memorie respectă relația :

$$t = (n + n^m_{1/2})/r^m_\infty \tag{1.20a}$$

iar operațiile vectoriale executate în unitatea aritmetică pipeline respectă relația :

$$t = (n + n^a_{1/2})/r^a_\infty \tag{1.20b}$$

Introducînd noua variabilă importantă f (*intensitatea computațională*), definită ca numărul operațiilor aritmetice în virgulă mobilă per referință la memorie şi exprimînd timpul mediu pentru o operație aritmetică vectorială ca :

$$\bar{t} = (n + \bar{n}_{1/2})/\bar{r}_\infty \tag{1.21a}$$

atunci, în cazul cînd operațiile de transfer cu memorie şi cele aritmetice nu se pot suprapune, găsim că :

$$\bar{r}_\infty = \frac{r^\infty_a}{1 + x^{-1}} = \hat{r}_\infty \ \text{pipe}(f/f_{1/2}) \tag{1.21b}$$

$$\bar{n}_{1/2} = \frac{n^m_{1/2} + n^a_{1/2}x}{1 + x} \tag{1.21c}$$

unde $\hat{r}_\infty = r^\infty_a$, $f_{1/2} = r^a_\infty/r^m_\infty$ şi $x = f/f_{1/2}$.

Astfel, raportul x determină gradul în care performanța medie a operațiilor aritmetice şi de acces combinate tind asimptotic spre performanța unității pipeline singure. Performanța maximă $\hat{r}_\infty$ definită la limita $f \to \infty$, este egală în acest model cu r^a_∞. Ecuația (1.21b) arată că modul cum este atinsă această asimptotă este identic cu cel în care performanța medie a unei singure unități pipeline, r, tinde spre asimptota ei, r_∞, ca o funcție de lungimea vectorului n, ca funcția pipeline pipe(x) (vezi ecuația (1.9a) şi figura 1.16). Deci, prin analogie cu $n_{1/2}$, introducem un nou parametru $f_{1/2}$ (*intensitatea performanței jumătate*), care este valoarea lui $f_{1/2}$ necesară pentru a atinge jumătate din performanța maximă. Parametrul $f_{1/2}$ este o proprietate numai a hard-ului calculatorului şi reprezintă o măsură față de care f (care este o funcție numai de algoritm şi aplicație) trebuie comparat, pentru a estima performanța medie, $\bar{r}_\infty$. Dacă $f = f_{1/2}$, performanța medie este jumătate din cea maximă posibilă (ca şi pentru $n_{1/2}$) ; dar dacă dorim atingerea a 90% din performanța maximă (0, $9r^a_\infty$) avem nevoie ca $f = 9f_{1/2}$. Ecuația (1.21c) arată cum variază $n_{1/2}$ pentru operații de transfer şi aritmetice combinate de la cel al memoriei pentru f mic, pînă la cel al unității pipeline pentru f mare. Fig. 1.21(a) şi (b) pre-

zintă măsurări ale lui $f_{1/2}$ pentru **FPS 164**, conectat printr-un canal la **IBM 4381** ca gazdă (Hockney 1987c). Măsurători similare au fost publicate pentru **FPS 5000** (Curington şi Hockney 1986).

Valoarea lui $f_{1/2}$ este determinată prin trasarea lui f/r_∞ versus f, care ar trebui să fie aproximativ liniar. Inversa pantei celei mai bune linii drepte este $\hat{r}_\infty$, iar intersecţia negativă cu axa f este $f_{1/2}$, deoarece prin rearanjarea ecuaţiei (1.21b) se obţine:

$$f/r_\infty = \hat{r}_\infty^{-1}(f + f_{1/2}) \qquad (1.21d)$$

Dacă, pe de altă parte, transferurile cu memorie pot avea loc în acelaşi timp cu execuţia operaţiilor aritmetice în pipeline, cele două tipuri de operaţii se pot suprapune. În acest caz, rezultă o expresie funcţională diferită, pe care o definim ca funcţia knee. Aceasta este funcţia rampă trunchiată:

$$\text{knee}\ (x) = \begin{cases} x, & \text{dacă}\ x < 1 \\ 1, & \text{dacă}\ x > 1 \end{cases} \qquad (1.22a)$$

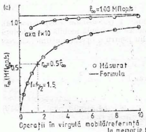
Performanţa operaţiilor aritmetice şi de transfer suprapuse este atunci:

$$r_\infty^- = \hat{r}_\infty^- \,\text{knee}_-(0{,}5f/f_{1/2}) \qquad (1.22b)$$

unde $\hat{r}_\infty = \hat{r}_\infty^a$; iar $f_{1/2} = 0{,}5 r_\infty^a / r_\infty^m$ (1.22c)

Astfel, prin suprapunerea operaţiilor de transfer, valoarea lui $f_{1/2}$ este înjumătăţită şi se reduce, în mod corespunzător, numărul operaţiilor aritmetice per referinţă la memorie necesare pentru a atinge un anumit procent din performanţa maximă.

În analiza de mai sus, dacă memoria şi unitatea aritmetică pipeline se află în acelaşi CPU, realizăm o analiză a unui calculator **SISD** sau **SIMD** cu limitarea traficului la memorie. Dacă, pe de altă parte, memoria şi unitatea pipeline se află în CPU diferite, realizăm o analiză a overheadului de comunicaţie la un calculator **MIMD**. În ambele cazuri parametrul hardware cheie este $f_{1/2}$ care, în acest model de calcul, este proporţional, cu



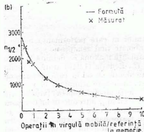


Fig. 1.21. a) Variaţia performanţei asimptotice, r_∞, ca o funcţie de numărul operaţiilor în virgulă mobilă per referinţă la memorie f, pentru un FPS 164 conectat via un canal la IBM 4351. b) Lungimea performanţei jumătate medii, $n_{1/2}$, ca o funcţie de numărul operaţiilor în virgulă mobilă per referinţă la memorie, f, pentru un FPS 164 conectat printr-un canal la IBM 4381

conţinute de toate variabilele de intrare, ieşire şi masive ale subrutinei, pentru toate instanţierile ei. În cazul acestui test, dacă n este lungimea vectoru- lui m = 3n (2 vectori de intrare, 1 vector de ieşire) şi s = 3nf deoarece operaţiile aritmetice se repetă de 3f ori.

Prin rearanjări se arată că :

$$r_\infty = \hat{r}_\infty \text{pipe } (f/f_{1/2}) ; \; s_{1/2} = \hat{s}_{1/2} \text{pipe } (f/f_{1/2}) \tag{1.25a}$$

$$\text{şi } r = r_\infty \text{pipe}(s/s_{1/2}) = \hat{r}_\infty \text{pipe } (f/f_{1/2}) \text{ pipe}(s/s_{1/2}) \tag{1.25b}$$

sau

$$r = \frac{\hat{r}_\infty}{1 + (f_{1/2}/f) + (\hat{s}_{1/2}/s)} \tag{1.25c}$$

unde $\hat{r}_\infty = t_a^{-1}$ $\hat{s}_{1/2} = t_0/t_a$ şi $f_{1/2} = t_t/t_a$ (1.25d)

Iată interpretarea ecuaţiei (1.25) : Principalii parametri hardware care determină comportarea calculatoarelor MIMD sînt t_0, t_t şi t_a. Aceştia influenţează performanţa programului prin rapoartele definite în ecuaţia (1.25d). Performanţa maximă este $\hat{r}_\infty$, inversul timpului consumat de operaţiile aritmetice. Valoarea maximă a lui $s_{1/2}\hat{s}_{1/2}$ este raportul între timpul de sincronizare şi cel consumat pentru execuţia operaţiilor aritmetice, numărul maxim posibil de operaţii aritmetice pierdute datorită sincronizării. Parametrul de comunicaţie $f_{1/2}$ este raportul dintre timpul necesar comunicaţiilor şi timpul necesar operaţiilor aritmetice, raportul dintre viteza de calcul şi viteza de comunicaţie (banda). Parametrii performanţei generale (r_∞, $s_{1/2}$) sînt mai mici decît valorile hardware maxime datorită întîrzierilor de comunicaţie, această variaţie se exprimă în ecuaţia (1.25a) prin funcţia pipeline.

Avînd calculaţi parametrii generali, performanţa medie actuală este furnizată de ecuaţia (1.25b), unde ultima expresie arată că rata maximă este micşorată datorită comunicaţiilor (prima funcţie pipeline), şi de dimensiunea inadecvată a granulaţiei (a doua funcţie pipeline). Trebuie reamintit că în ecuaţia (1.25b) $s_{1/2}$ este o funcţie de f prin ecuaţia (1.25a). Variaţia reală a vitezei medii cu f şi s se observă mai clar în ecuaţia (1.25c), unde se folosesc numai parametrii hardware reali. În ecuaţia (1.25c) primul termen de la numitor provine de la timpul de calcul aritmetic, al doilea de la timpul de comunicaţie iar al treilea de la timpul de sincronizare.

Fig. 1.22(a) şi (b) prezintă măsurători ale parametrilor de performanţă gernerală (r_∞, $s_{1/2}$) conform ecuaţiei (1.23) pentru testul ($\hat{r}_\infty$, $\hat{s}_{1/2}$, $f_{1/2}$) aplicat sistemului de calcul IBM/CAP (vezi §2.6.9) unde la un calculator gazdă IBM 4381 sînt conectate 10 FPS 164 (Hockney 1987d). Deoarece $f_{1/2} = 2$, curbele superioare pentru f=100 corespund lui $\hat{r}_\infty$ şi $\hat{s}_{1/2}$, iar cele inferioare prezintă degradarea performanţelor ce intervine cînd f este mic. Curbele simbolizate cu 1 şi 2 se obţin folosind ecuaţia (1.25) pentru următorii parametri :

$$\begin{aligned} \hat{r}_\infty &= 1{,}08p \text{ Mflop/s} \\ \hat{s}_{1/2} &= 1{,}08p \; (2500 + 4777p) \text{ flop} \\ f_{1/2} &= 2{,}02 \text{ flop/ref.} \end{aligned} \tag{1.26}$$

raportul performanței aritmetice la rata transferului. În lucrările lui Lint și Agerwala (1981) și Lee, Abu-Sufah și Kuck (1984) sînt prezentate alte analize ale degradării performanțelor datorită problemelor ridicate de comunicații.

(iv) *Testul* ($\hat{r}_\infty$, $\hat{s}_{1/2}$, $f_{1/2}$)

În cursul analizei de mai sus am considerat overhead-urile de sincronizare pentru f constant (partea (ii)), ca și efectele întîrzierilor datorate comunicațiilor fără sincronizare (partea (iii)). În continuare vom considera un test mai general care include ambele efecte și, în particular, prezintă variația lui $s_{1/2}$ cu f.

Vectorul **A** va fi calculat ca produsul vectorial element cu element al vectorilor **B** și **C**, prin împărțirea volumelor de calcule cît mai egal între cele p procesoare. Inițial, vectorii **B** și **C** se află în memoria sistemului principal (oricare ar fi acesta), iar vectorul rezultat **A** se va asambla în aceeași memorie. Astfel, fiecăruia din cele p procesoare i se va trimite o parte a vectorilor **B** și **C**, va executa o înmulțire parțială și va trimite porțiunea din vectorul **A** rezultat, memoriei principale. Job-ul este încheiat numai cînd toate procesoarele au returnat memoriei principale vectorii parțiali calculați. Cu alte cuvinte, după p operații de calcul în program trebuie stabilit un punct de sincronizare (prin includerea, de exemplu, a unei instrucțiuni ,,barieră''), care va fi inclus în măsurarea timpului de execuție. Pentru a varia raportul între operațiile de calcul și cele de comunicație, fiecare procesor repetă înmulțirea vectorilor parțiali **B** și **C** de 3f ori. Se păstrează definirea anterioară a lui f, deoarece există trei operații de I/E (input **B**, **C**; output **A**). Cînd $f = 1/3$, operația vectorială se execută o dată, așa cum s-a prezentat în (ii).

Timpul total de execuție a testului este:

$$t = r_\infty^{-1}(s + s_{1/2}) \text{ rezultînd } r = r_\infty \, \mathrm{pipe}(s/s_{1/2}) \qquad (1.23)$$

Vom separa contribuția părților de calcul, comunicație și sincronizare și vom considera situația cînd operațiile aritmetice nu se pot suprapune în timp cu cele de comunicație. Atunci,

$$t = t_0(p) + t_t(p)m + t_a(p)s \qquad (1.24a)$$

$$t = t(\textit{inițial și sincronizare}) + t(\textit{comunicații}) + t(\textit{calcul}) \qquad (1.24b)$$

unde cei trei termeni sînt identificați cu inițializarea, comunicațiile și, respectiv, calculul. Termenii din ecuație se explică după cum urmează s este numărul total de operații în virgulă mobilă (flop) executate de toate procesoarele; m este numărul cuvintelor ce se transferă prin operații de I/E în cadrul segmentului de lucru (vezi mai jos); f este egal cu s/m și dă numărul de operații în virgulă mobilă pe cuvînt transferat; $t_a(p)$ este timpul mediu pentru execuția unei operații în virgulă mobilă, $t_0(p)$ este timpul pentru job-ul nul ($s = m = 0$); t (p) este timpul mediu pentru o operație de I/E.

Variabila *m* cuantifică comunicația realizată de un segment de lucru cu restul programului. Dacă, așa cum se întîmplă de obicei, corpul segmentului de lucru este scris ca o subrutină, reprezintă numărul cuvintelor

Acești parametri arată că pentru ca o problemă să poată fi rezolvată eficient pentru același sistem, trebuie să aibă $f = 8$ flop/ref de unde $f/f_{1/2} = 4$ și o performanță asimptotică de 80% din cea maximă. Pentru 10 procesoare $s_{1/2}$ ar fi de aproximativ 400 Kflop și pentru o eficiență de 80% ar fi necesară o dimensiune a granulației de $4s_{1/2} = 1{,}6$ Mflop. Aceasta este semnificația numerică a aserțiunii că acest sistem este slab conectat și deci indicat numai pentru probleme ce au paralelism cu granularitate mare (de ordinul 1 Mflop). Din fericire, multe probleme de fizică satisfac acest criteriu (Fox și Otto 1984).

(v) *Performanța programului*

Secțiunile anterioare au tratat numai determinarea timpului de execuție pentru un singur segment de lucru care poate fi distribuit la p procesoare. Cînd se consideră un program complet, cu mai multe segmente de lucru, trebuie să se observe că unele pot conține cod ce trebuie executat secvențial pe un singur procesor, pentru a se obține rezultate corecte. Să împărțim atunci timpul de execuție al programului secvențial original, T_1, în două părți :

$$T_1 = t_{ser} + t_{par} \qquad (1.27)$$

unde t_{ser} este timpul necesar execuției părții secvențiale din cod, iar t_{par} cel necesar pentru execuția părții paralele.

Cînd programul este paralelizat se obține timpul de execuție paralelă cel mai bun :

$$T_p = t_{ser} + t_{par}/p \qquad (1.28)$$

unde numai al doilea termen se reduce prin execuție paralelă. Am presupus, desigur, că timpii de sincronizare și comunicare sînt neglijabili. Dacă definim performanța programului R(p) ca inversul timpului de execuție, atunci :

$$R(p) = T_p^{-1} = (t_{ser} + t_{par})^{-1} = R_\infty[1 + (p_{1/2}/p)]^{-1} \qquad (1.29a)$$

care poate fi rescrisă

$$R(p) = R_\infty \mathrm{pipe}(p/p_{1/2}) \qquad (1.29b)$$

unde $R_\infty = t_{ser}^{-1}$ și $p_{1/2} = t_{par}/t_{ser}$ (1.29c)

Astfel, rata asimptotică R_∞ este inversă timpului necesar execuției acelei părți a programului care nu poate fi paralelizată. Această situație intervine în model cînd numărul procesoarelor $p \to \infty$, ceea ce reduce timpul de execuție a părții paralelizate la zero. Observăm că forma funcțională de atingere a ratei asimptotice este din nou cea a unei funcții pipeline pipe (x), iar $p_{1/2}$, ca de obicei, este numărul procesoarelor necesare pentru atingerea a jumătate din performanța asimptotică.

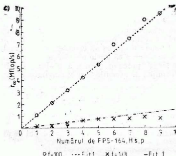
Parametrul cel mai folosit pentru compararea performanțelor algoritmilor este *accelerarea* (speed-up), Sp (Kuck 1978), definită cu :

$$S_p = T_1/T_p = R(p)/R_{(1)} = S_\infty[1 + (p_{1/2}/p)]^{-1} = S_\infty \mathrm{pipe}(p/p_{1/2}) \qquad (1.30a)$$

unde T_1 şi T_p sînt timpii de execuţie ai algoritmilor pe un procesor, respectiv p procesoare. Se obţine:

$$S_\infty = R_\infty/R(1) = (1 + p_{1/2}) = T_1/t_{ser}$$

=(fracţia din timpul de execuţie al programului iniţial *neparalelizată*)$^{-1}$ (1.30n)



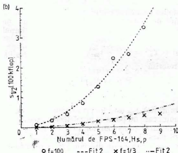


Fig. 1.22 (a) şi (b) prezintă măsurători ale parametrilor de performanţă generală (r_∞, $s_{1/2}$) conform ecuaţiei (1.23) pentru testul ($\hat{r}$, $s_{1/2}$, $f_{1/2}$) aplicat sistemului de calcul IBM ICAP (vezi §2.5.9) la care sînt conectate 10 FPS 164 la un calculator gazdă IBM 4381 (Hochney 1987 d).

Ambele ecuații (1.29b) și (1.30a) sînt expresii ale legii Amdahl (Amdahl 1967), în sensul că performanța sau accelerarea nu o poate depăși pe cea obținută cînd părțile vectorizate sau paralelizate se execută în timp zero. Rata maximă este determinată de timpul de execuție a operațiilor nevectorizate sau neparalelizate ale programului.

Se constată că multe programe paralele respectă dependența funcțională a ecuației (1.29b), cel puțin pentru valori mici ale lui p. În practică, cu cît p crește, sincronizarea și alte overhead-uri cresc rapid cu p, cu rezultatul că există de obicei un maxim al execuției programului, urmată de o reducere. Această comportare respectă funcția :

$$R(p) = R_\infty\{1 + (p_{1/2}/p)[1 + (1/(n-1))(p/\tilde{p})^n]\}^{-1} \qquad (1.31a)$$

În această formulă overhead-ul de sincronizare este reprezentat de factorul dintre acolade, iar rapiditatea cu care crește odată cu p este determinată de n, *indexul de sincronizare*. Evident, este de dorit să fie cît mai mic posibil.

Mărimea overhead-ului de sincronizare este determinată de $\tilde{p}$, unde valori mari înseamnă overhead mic. Valoarea acestor parametri variază considerabil pentru diversele sisteme software. Performanța maximă se obține cînd $p = \tilde{p}$ și are valoarea :

$$\tilde{R} = R_\infty[1 + (n/(n-1))(p_1/\tilde{p}_{/2}/)]^{-1} \qquad (1.31b)$$

Ca un exemplu practic de paralelizare am considerat un program de simulare a mișcării electronilor în semiconductori, denumit **FET6**, convertit pentru execuție paralelă pe **IBM/CAP** la centrul IBM ECSEC din Roma (Hockney 1987c). Programul calculează noile poziții și viteze a aproximativ 10000 electroni, pas cu pas. El este descris în lucrarea Hockney și Eastwood (1981), capitolul 10. Fiecare pas constă din parcurgerea a trei etape: *atribuirea sarcinii, aflarea potențialului* și a *accelerației*. În timpul etapei de accelerare, deplasarea tuturor electronilor este independentă, astfel că această parte a programului poate fi paralelizată prin atribuirea a 1/p din electroni fiecăruia din cele p procesoare. Deoarece această etapă consumă timpul cel mai mare, celelalte părți nu sînt modificate și formează partea secvențială a programului.

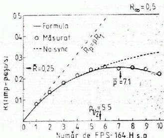


Fig. 1.23. Performanța absolută a programului FET6 executat pe 1 pînă la 10 calculatoare FPS 164. Performanța atinge un maxim $\tilde{R}$ pentru $\tilde{p}$ procesoare. Valorile măsurate respectă ecuația (1.31a) cu $R_\infty = 0.5$, $p_{1/2} = 5.5$, $\tilde{R} = 0,25$, $\tilde{p} = 7.1$ și $n = 5$. Dacă overhead-ul de sincronizare ar putea fi eliminat, performanța ar urma curba întreruptă cu asimptota la R_∞.

Fig. 1.23 arată măsurătorile de performanța ale programului FET6, executat pe mai multe FPS 164. Timpul de execuție al programului pa-

ralelizat respectă ecuația (1.31a), în domeniul $1 \leqslant p \leqslant 10$ cu următorii parametri (în paranteze se dau rapoartele accelerare la R(1) echivalente):

$$R_\infty = 0{,}5 \text{ pași pe secundă } (S_\infty = 6{,}2) \quad (1.32)$$

$$p_{1/2} = 5{,}5 \quad \tilde{p} = 7{,}1 \quad n = 5$$

Valoarea maximă a performanței este, conform ecuației (1.31b):

$$\tilde{R} = 0{,}25 \text{ pași pe secundă } (\tilde{S} = 3{,}1) \quad (1.33)$$

Iată interpretarea semnificației acestor parametri: valoarea $\tilde{p} = 7{,}1$ înseamnă că programul paralelizat nu poate fi utilizat cu folos mai mult de 7 procesoare, și în acest mod problema este rezonabilă pentru execuția pe sistemul /**CAP** cu 10 procesoare. Valorile ($\tilde{R}$, $\tilde{p}$, n) depind de overhead-ul de sincronizare. Dacă ar putea fi redus substanțial prin folosirea unui hardware de sincronizare sau îmbunătățiri software, efectul ar fi creșterea $\tilde{p} \to \to \infty$ și $\tilde{R} \to R_\infty$. Deoarece sincronizarea este realizată de către calculatorul gazdă, ar fi necesar ca acesta să fie mai rapid. Astfel, R_∞ este performanța maximă posibilă pentru program, cînd timpul de sincronizare este neglijabil. 90% din această performanță poate fi atinsă dacă numărul procesoarelor $p = 9p_{1/2} = 50$, deci mai multe decît sînt instalate în sistemul /CAP de la Roma.

În fig. 1.23, performanța ideală ar urma linia pR_1, de-a lungul căreia performanța este de p ori mai mare decît pentru un procesor (idealul accelerării liniare). Deoarece numai o parte a programului poate fi paralelizată, legea Amdahl limitează performanțele la mai puțin decît $R_\infty = 0{,}5$ pași/sec. În acest mod, zidul Amdahl este un plafon Amdahl.

În absența overhead-ului de sincronizare, performanța ar respecta funcția pipeline cu $p_{1/2} = 5{,}5$ procesoare, lucru arătat de linia întreruptă. După depășirea numărului de 5 procesoare, overhead-ul de sincronizare reduce rapid performanța care atinge un maxim la $\tilde{p} = 7{,}1$ procesoare cînd performanța este $\tilde{R} = 0{,}25$, o accelerare de numai $\tilde{S}_7 = 3{,}1$ în comparație cu performanța unui singur procesor.

Comportarea performanței programului în funcție de numărul de procesoare este tipică pentru sistemele de calcul MIMD. Scopul este de a produce hard și soft astfel încît R_∞ și $\tilde{p}$ să fie cît mai mari și $p_{1/2}$ cît mai mic.

2.

CALCULATOARE PIPELINE

2.1 SELECȚIE ȘI COMPARAȚIE

În acest capitol vom descrie în detaliu arhitectura, tehnologia și performanța mai multor calculatoare pipeline. Pentru început vom discuta ultimele modele ale seriei **CRAY** de supercalculatoare, **CRAY X-MP** și **CRAY-2**. De departe acestea au avut succesul comercial cel mai mare, mai mult de 88 exemplare din modelele CRAY-1, CRAY-1S, -1M și X-MP fiind instalate pînă în anul 1985. În cursul anilor 1984/5 au fost instalate 3 prototipuri CRAY-2 cu un CPU, iar pînă la sfîrșitul anului 1987 au fost instalate 13 sisteme cu 4 CPU.

A doua serie de calculatoare descrisă, CDC CYBER 205, este de asemenea bine cunoscută și datorită vînzărilor de aproximativ 30 bucăți pînă în 1985. Arhitectura sa are o istorie lungă, inițiată cu **CDC STAR 100**, calculator proiectat la mijlocul anilor '60 și diferă suficient de mult de cea a calculatoarelor **CRAY**. Viitorul acestei linii este reprezentat de **ETA**[10] care poate fi descris ca un **CYBER 205** cu 8 CPU cuplate la o memorie comună, implementat în tehnologie CMOS și răcit cu hidrogen lichid. Cu acest model, arhitectura calculatorului **CYBER 205** va fi de asemenea de mare interes.

Cele 3 calculatoare vectoriale japoneze, FUJITSU VP 100/200, **HITACHI S-810/10** și **20** și **NEC SX1/SX2** au apărut mai tîrziu și, în mod evident, datorează mult arhitecturii calculatorului **CRAY** și **CYBER 205**. Deoarece sînt similare, le vom trata împreună, comparativ.

Ultima serie de calculatoare pipeline pe care o discutăm este Floating Point Systems 164 și 164/MAX. Aceste calculatoare au aproape aceeași arhitectură cu precedentele FPS-120B și FPS-100, din care au fost instalate peste 5500 exemplare pînă în anul 1985. Ele au avantajul că se situează în gama de prețuri a multor universități și întreprinderi industriale și, cînd sînt programate optim, pot furniza rapoarte preț/performanțe excepționale. De un interes special se bucură acceleratorul matricial, sau MAX, o placă din care pot fi cuplate pînă la 15 la un FPS-164. Fiecare din ele este echivalentă cu 2 CPU FPS-164 ce lucrează sub controlul unui flux de instrucțiuni unic. Pentru probleme cu matrici, se ating performanțe în domeniul supercalculatoarelor la prețul unui minicalculator. O altă direcție interesantă de dezvoltare este cea inițiată de Clementi la IBM Kingston

și Roma, ca și de Wilson la Universitatea Cornell, care conectat aproximativ 10 FPS-164 la un calculator gazdă, rezultînd astfel un sistem MIMD economic pentru rezolvarea a numeroase probleme științifice. De aceea, arhitectura calculatorului FPS-164 ne interesează foarte mult.

În comparațiile ce vor urma, vom descrie : *structura fizică*, pentru a crea o idee despre dimensiunea și forma mașinii, *arhitectura* de ansamblu, pentru a evalua ierarhia memoriei, căile de date, vitezele de transfer și elementele de calcul, *tehnologia*, *setul de instrucțiuni* pentru a-i evalua bogăția, *soft-ul* disponibil, și apoi în final, se apreciază *performanța* prin măsurători ale lui r_∞ și $n_{1/2}$ obținute, cînd a fost posibil, prin execuția unor programe pe calculatoarele respective.

2.2 CRAY X-MP ȘI CRAY-2

Atît **CRAY X-MP**, cît și **CRAY-2**, sînt produse de CRAY Research Inc. la Chippewa Falls, Wisconsin, SUA. Ele au derivat de la CRAY-1, proiectat de Seymour Cray și instalat prima oară la Los Alamos Scientific Laboratory în 1976 (Auerbach 1976b, Hockney 1977, Russell 1978, Dungworth 1979, Hockney și Jesshope 1981). Apariția lui CRAY X-MP care este versiunea multiprocesor a lui CRAY-1, se datorează lui Steve Chen și echipei sale (Chen 1984) de la Chippewa Falls. În 1982 a fost anunțată o versiune cu 2 CPU, iar în 1984 o versiune cu 4 CPU. Informațiile privind CRAY-X-MP provin în primul rînd din manualele de referință CRAY X-MP Computer Systems (CRAY 1982, 1984a, 1984b). Următorul produs din serie este CRAY Y-MP anunțat în 1987. Se anticipează că acesta va avea o perioadă a orologiului de 4—5 ns și că va consta din pînă la 16 CPU și 32 Mw de memorie comună, plus memorie secundară cu capacitatea de 1 Gw.

Pe de altă parte, CRAY-2 a fost dezvoltat de Seymour Cray ca un proiect separat la Chippewa Falls. El folosește tehnologie mai avansată și un nou mod de răcire. Astfel, se pot folosi orologiile cele mai rapide în momentul de față, c1988 în calculatoarele produse comercial, cu o perioadă de 4,1 ns CRAY-2 a fost anunțat ca produs în 1985 CRAY 1985).

2.2.1. Structura fizică

Caracteristica cea mai izbitoare a calculatoarelor CRAY este dimensiunea lor redusă. Aceasta este bine ilustrată în fig. 2.1 care prezintă un sistem CRAY X-MP. În centrul este chiar CRAY X-MP care este asamblat în aceeași structură cilindrică ca și calculatoarele anterioare CRAY-1 și CRAY-1S. În stînga se află sfertul de cilindru cu memoria SSD (solid state device), denumită uneori ca discul solid state, iar la dreapta, în alt sfert de cilindru, se află subsistemul de I/E (IOS).

CRAY X-MP constă dintr-o coloană cilindrică centrală de 1,5 m diametru și 1,9 m înălțime, înconjurată de o banchetă circulară ce face ca diametru să fie la bază de aproximativ 2.7 m. Coloana centrală este

împărţită în trei segmente de 90°, fiecare formată din 4 coloane ce au la rîndul lor cîte 144 module cu circuite. Fiecare modul este format dintr-o pereche de plăci cu circuite montate pe părţile opuse ale unei plăci de cupru, care face un contact termic bun cu barele de răcire din aluminiu,

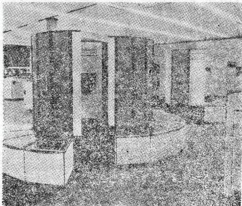


Fig. 2.1 O instalare CRAY X-MP tipică ce constă din calculator, în centru, subsistemul de I/E la dreapta şi dispozitivul solid state (SSD) la stînga. (Fotografie furnizată de Cray Research Inc.)

verticale, lucru arătat în fig. 2.2. Aceste bare de răcire sînt evidente în coloana verticală din fig. 2.2. Prin conducte de oţel introduse în barele de răcire circulă freon, care menţine temperatura de 21 grade. Temperatura plăcii de cupru este menţinută la 25 grade, iar cea a modulelor între 48 şi 54 grade Celsius. Bancheta de la bază maschează sursele de alimentare, şi conductele sistemului de răcire. Două compresoare de 25 tone aflate în exteriorul sălii calculatorului completează sistemul de răcire. Un generator extern de 175 KVA asigură tensiunea de 208 V trifazat la 400 cicli. Consumul total de putere este de 128 kW, din care multă trebuie eliminată prin sistemul de răcire. CRAY X-MP ocupă 15 m² şi cîntăreşte 5,25 t. Subsistemele IOS şi SSD ocupă fiecare 5 m² şi cîntăresc 1.5 t.

Fig. 2.3 prezintă un modul cu 2 straturi. Se ataşează 2 plăci cu circuite la fiecare din cele 2 plăci de cupru, care fixate în mod rigid formează o structură tri-dimensională de 4 plăci. Între cele 4 plăci cu circuite sînt realizate conexiuni electrice. Apoi, modulul cu 4 plăci glisează în canelură, cum se vede în fig. 2.2.

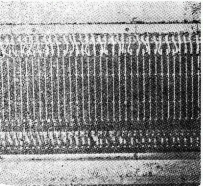


Fig. 2.2. O coloană verticală de plăci cu circuite ale calculatorului CRAY X-MP. Plăcile sînt atașate în perechi la sistemul de răcire cu freon. (Fotografie furnizată de Cray Research Inc).

Fig. 2.3 O placă cu circuite a calculatorului CRAY X–MP Fiecare placă are capacitatea maximă de 144 capsule. (Fotografie furnizată de Cray Research Inc.)

CRAY X-MP își atinge în parte, performanța datorită aranjării compacte a plăcilor cu circuite care determină folosirea unor fire scurte și timpi de propagare mici. Acest lucru poate fi văzut în figura 2.4 care prezintă modul de aranjare al modulelor în calculatorul X-MP/48. În total există 12 coloane, ce formează un arc de 270 grade. Cele 4 coloane centrale conțin cele 4 CPU, iar cele 4 coloane din stînga, respectiv din dreapta, conțin cei 8 Mw de memorie internă.

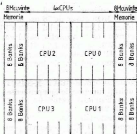


Fig. 2.4 (a) Modul de organizare fizică a memoriei și unităților centrale în interiorul coloanei centrale a calculatorului CRAY X–MP/48. (b) Alocarea pozițiilor plăcilor cu circuite fiecărei unități, în cadrul unui sfert (de exemplu, CPU 1)

a)

Real-time clock
A and B Address registers
Address add
Address multiply
Error correction
Vector register control
8 Vector registers
Vector add

Instruction buffer
Shared registers
Program counter
Vector logical
Functional unit to vector register switch
VL register
Vector register to funct onal unit switra
Vector shift

Error correction
S and T Scalar registers
Scalar shift
Scalar add
Scalar shift
S and T Scalar registers
Floating add

I/O Channels
Reciprocal table
Reciprocal approximation
Floating-point multiply

8 Memory banks
128 Kword per bank
4 CPUs to memory switch

Memory to 4 CPUs switch
8 Memory banks

2.2.2. Arhitectura

Arhitectura de ansamblu a calculatorului **CRAY X-MP** poate fi descrisă ca una, două sau 4 CPU de tip CRAY-1 care partajează o memorie comună de pînă la 8 Mw. Modelul inițial, cu 2 CPU, introdus în 1982 ocupă 3/4 dintr-un cilindru sub forma a 12 coloane, așa cum se arată în fig. 2.1. Folosirea circuitelor cu densitate mai mare a permis în 1984 ca modelele cu 2 CPU sau 1 CPU să ocupe numai 6 coloane, în timp ce modelul cu 4 CPU utiliza complet cele 12 coloane. Modelul cu 1 CPU poate fi livrat cu 1,2 sau 4 Mw de memorie MOS static de 76 ns, organizată în 16 blocuri (1 sau 2 Mw) sau 32 blocuri (4Mw); modelul cu 2 CPU și 2 sau 4 Mw bipolară ECL de 38 ns organizată în 16 respectiv 32 blocuri. Modelul cu 4 CPU are 8 Mw de memorie ECL organizată în 64 blocuri. Ultimul calculator este denumit CRAY X-MP/48 unde prima cifră indică numărul unităților centrale, iar al doilea numărul Mw de memorie comună.

Structura bloc din fig. 2.5 reprezintă sistemele CRAY X-MP/22 și X-MP/24. Fiecare CPU are 13 sau 14 unități funcționale independente ce lucrează cu registre (modelul X-MP/48 are 2 unități vectoriale). Ca și la CRAY-1, există 8 registre pentru adrese de 24 biți (A0,...,A7), 8 registre scalare de 64 biți (S0,... S7) și 8 registre vectoriale (V0,...,V7), fiecare avînd posibilitatea de a memora 64 elemente pe 64 biți. Registrele pentru adrese au propriile unități pipeline pentru operațiile de adunare și înmulțire cu întregi, în timp ce registrele scalare și vectoriale au propriile unități pipeline pentru adunare întreagă, deplasare, operații logice și numărare. Operațiile în virgulă mobilă se execută în 3 unități pipeline, pentru înmulțire, adunare și aproximare a inversului (RA), care primesc operanzii fie din registrele scalare, fie vectoriale. Să observăm că operațiile în virgulă mobilă scalare și vectoriale nu pot fi executate simultan, ca la acele calculatoare care au unități scalare și vectoriale separate (de ex. CYBER 205). Un registru de 7 biți (vector length — VL) specifică numărul elementelor (pînă la 64) implicate într-o operație vectorială, iar un registru pe 64 biți (vector mask — VM) are un bit pentru fiecare element al unui registru vectorial și este folosit pentru mascarea anumitor elemente, în sensul neparticipării la o operație vectorială.

Pentru a se executa operații aritmetice, trebuie ca întîi datele să fie transferate din memoria comună în registre. Aceste operații se execută direct sau, în cazul registrelor S și A prin transferuri pe bloc în registrele bufer T și B care pot memora cîte 64 elemente. Transferul la registrele S și A se execută în modul un cuvînt la fiecare două perioade de ceas, iar la registrele T și B cîte 3 cuvinte la o perioadă de ceas. Transferul între registrele vectoriale și memoria comună se realizează pe 3 căi de date de 64 biți. Două căi furnizează date registrelor vectoriale din memoria comună, iar al treilea transferă rezultatul din registrul vectorial în memoria comună. Deoarece căile de date sînt separate, într-o perioadă de ceas, se pot transfera 2 argumente și un rezultat, ceea ce înseamnă o rată de transfer de 315 Mw/s per CPU. O mașină cu 4 CPU are, deci o lărgime a accesului la memorie de 1.2 Gw/s. Dacă nu se produc conflicte de acces la memorie (vezi următorul paragraf), se pot executa operații cu memorie cu această

viteză. Deoarece, cînd este încărcat fiecare pipeline în virgulă mobilă necesită două argumente și produce un rezultat la fiecare perioadă de ceas, lărgimea de bandă este exact necesarul pentru realizarea unei operații în virgulă mobilă cu date din memorie. Dacă, simultan, mai trebuie să fie active și alte unități pipeline, rezultatele intermediare trebuie memorate

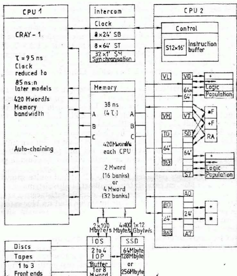


Fig. 2.5 Diagrama bloc a lui CRAY X—MP/2, care prezintă două CPU, memoria și căile principale de date. Literele reprezintă abrevierile Cray pentru registrele mașinii (de exemplu, A, B, S, T și V). Ele sînt urmate de un număr zecimal care arată numărul de registre. 64' reprezintă numărul de biți ai registrelor. SB, ST, VM și VL sînt alte registre. 'F' la sfîrșit semnifică operații în virgulă mobilă.

în registre. În acest mod este proiectată arhitectura orientată-registru a mașinilor CRAY. Avantajul registrelor este că, atunci cînd sînt folosite optim, numărul acceselor la memorie se micșorează și se pot atinge viteze mari de calcul pentru buclele interioare. Costul este complexitatea mai mare a programării necesare pentru atingerea utilizării optime a registrelor, indiferent dacă se realizează de către compilator sau prin codificare manuală în limbajul de asamblare.

Pentru fiecare CPU există 4 porturi în memoria comună, dintre care unul (etichetat de I/E) este rezervat pentru subsistemul de intrare/ieşire şi SSD. Celelalte trei porturi (etichetate A, B, C) sînt cele folosite pentru operaţiile cu registre. Toate operaţiile de transfer cu registrele A şi B folosesc portul C. Transferurile de blocuri de la memoria comună la registrele B folosesc portul A, iar la registrele T folosesc portul B. Operaţia inversă de la registrele B şi T foloseşte portul C. Transferurile între memoria comună şi registrele vectoriale folosesc toate cele 3 porturi : porturile A şi B pentru a încărca registrele cu operanzi din memorie şi portul C pentru operaţia inversă. Astfel se realizează trei accese la memorie într-o perioadă de ceas. Cu 4 porturi şi pînă la 4 procesoare se accesează pînă la 64 blocuri de memorie, sistemul de interconectare a memoriei la CRAY X-MP este complex. Cheung şi Smith (1984) au analizat sistemul şi au propus unele îmbunătăţiri. În continuare vom descrie sistemul X-MP/24 cu 2 CPU şi 32 blocuri de memorie. Extensia la mai multe CPU blocuri urmează aceeaşi strategie.

Memoria este împărţită în 4 secţiuni, fiecare cu 8 blocuri de memorie a 128 Kw. (Cel mai mare X-MP/48 are 16 blocuri pe secţiune, cu fiecare 4 blocuri consecutive în aceeaşi secţiune). Există o cale de date independentă denumită *linie* (line), de la fiecare CPU la fiecare secţiune de memorie. Cele 4 linii ale fiecărui CPU sînt conectate cu un comutator în cruce la căile A, B, C şi de I/E ce vin de la unităţile funcţionale. Adresarea este întreţesută peste secţiuni, astfel că cei mai puţini semnificativi 2 biţi ai adresei dau numărul secţiunii, iar cei mai puţini semnificativi 5 biţi dau numărul blocului. Cînd este accesată, o linie este ocupată pentru o perioadă de ceas, iar un bloc pentru 4 perioade de ceas.

Conflictele de acces pot avea loc la X-MP în mai multe situaţii. *Conflictele de acces la bloc*, cu care sîntem familiari la alte sisteme, au loc cînd un bloc este accesat în timp ce este încă în curs de satisfacere a unei cereri anterioare. Un *conflict de simultaneitate* intervine cînd un bloc este accesat simultan pe linii independente de către CPU diferite. Această situaţie se rezolvă prin atribuirea unei priorităţi diferite unităţilor centrale la fiecare 4 perioade. În sfîrşit există un *conflict pe linie* (line conflict) în cadrul aceluiaşi CPU, care intervine cînd două sau mai multe din căile A, B, C sau de I/E solicită un acces la aceeaşi secţiune de memorie în acelaşi ciclu maşină. Conflictele de linie sînt rezolvate prin atribuirea priorităţii unei referinţe vectoriale. Dacă ambele referinţe au aceeaşi paritate, primul acces solicitat este prioritar. Dacă intervin conflicte cînd se accesează vectori, element cu element, ele trebuie rezolvate şi se înserează stări wait. Aceasta înseamnă că intervalul de timp între referinţele elementelor vectorilor nu este previzibil sau constant. Deoarece aceste elemente alimentează unităţile pipeline, diverse nivele ale lor pot fi lipsite de operanzi. Aceste goluri (sau „bubbles") generează o degradare a vitezei de calcul aritmetice medii, funcţie de modul de desfăşurare a acceselor la memorie. Deoarece conflictele de acces la memorie pot depinde de activitatea unităţilor centrale care nu sînt controlabile de către programator, este probabil că degradarea să fie imprevizibilă. Totuşi, în cazul cel mai rău, a 4 CPU care execută 3 accese la memorie într-o perioadă de ceas la acelaşi bloc, fiecare CPU ar primi accesul la fiecare 16 perioade în comparaţie cu o rată maximă de 3 accese în fiecare perioadă, ceea ce duce la o degradare teoretică maximă a lărgimii de bandă la memorie printr-un factor de 48. Cheung şi

scalare lucrează exclusiv cu date pe 64 biţi aflate în registrele S, iar rezultatele, cu excepţia numărului de biţi (population count) sînt returnate în registrele S. Adunarea întreagă se realizează în complement faţă de 2. Deplasările pot fi executate atît cu registrul S simplu (2 perioade de ceas), cît şi cu 2 registre S concatenate (3 perioade de ceas). Unitatea logică, ce execută operaţii bit-cu-bit cu date pe 64 biţi, este parte a modulelor ce conţin registrele S. Pentru că datele nu părăsesc modulele unde se află registrele S, funcţia poate fi executată într-o singură perioadă de ceas. Unitatea de numărare a biţilor numără biţii cu valoarea 1 (în 4 perioade de ceas) sau numărul de zerouri ce precedă primul bit 1 al operandului (3 perioade de ceas). Rezultatul pe 7 biţi este returnat unui registru A.

Unităţile vectoriale folosesc operanzi ai unei perechi de registre V sau ai unui registru V şi unui registru S. Rezultatul este depus într-un registru V sau registru mască. În cursul fiecărei perioade de ceas se transmit unităţii funcţionale perechi succesive de operanzi. Rezultatul părăseşte unitatea 1 perioade de ceas mai tîrziu. După aceea, rezultatele se obţin la fiecare perioadă de ceas. Numărul elementelor procesate de o operaţie vectorială este egal cu cel memorat în registrul de 7 biţi a lungimii vectorului (VL). Elementele se numerotează cu 0, 1, 2 . . .Unele operaţii vectoriale sînt controlate de registrul de mascare vectorial (VM). Bitul n al măştii corespunde elementului n al registrului vectorial. Registrul VM este folosit de instrucţiuni care selectează anumite elemente ale vectorului, sau combină 2 vectori în unul singur (vezi §2.2.4). Unitatea de adunare vectorială a întregilor execută sume şi scăderi între numere pe 64 biţi reprezentate în complement faţă de 2. Unitatea de deplasare vectorială deplasează conţinuturile pe 64 biţi ale elementelor vectorului sau pe 128 biţi ale perechilor de elemente vectoriale adiacente. Numărul de deplasări este indicat prin instrucţiune sau un registru A. Unitatea logică vectorială execută operaţii logice bit-cu-bit cu elemente ale unui vector şi creează conţinutul registrului de mascare, VM.

Cele trei unităţi funcţionale în virgulă mobilă execută operaţii în virgulă mobilă atît cu scalari, cît şi cu vectori. De aceea, argumentele şi rezultatele se află în registrul S sau V. Numărul reprezentat în virgulă mobilă are o mantisă de 48 biţi, ce furnizează o precizie de aproximativ 14 cifre zecimale şi un exponent pe 16 biţi, ce defineşte un domeniu de valori normalizate între 10^{-2500} şi 10^{+2500}. Unităţi separate execută adunarea, înmulţirea şi împărţirea. Împărţirea se realizează cu aproximări reciproce, iar înmulţirea cu o procedură iterativă. Astfel, operaţia de împărţire poate fi realizată în manieră pipeline.

Algoritmul de calcul al raportului scalar S1/S2 este implementat cu 4 instrucţiuni maşină ce realizează următoarele operaţii:

unitatea RA	$S3 = 1/S2$	aproximare a inversului	(2.2a)
unitate de înmulţire	$S4 = (2 - S3*S2)$	iteraţie inversă	(2.2b)
	$S5 = S1*S3$	multiplicare	(2.2c)
	$S6 = S4*S5$	multiplicare	(2.2d)

concurent toate, se împart în 4 grupe. Perioada ceasului, τ, era la primele modele CRAY X-MP de 9.5 ns. La modelele ulterioare a fost redusă la 8.5 ns. În continuare vom folosi valoarea de 9.5 ns.

Unitatea funcțională	(ns)	Nr. de perioade = 1
Unitățile de adresare (24 biți)		
(1) adunare întreagă	19	2
(2) înmulțire întreagă	38	4
Unitățile scalare (64 biți)		
(3) adunare întreagă	28.5	3
(4) deplasare	19 sau 28.5	2 sau 3
(5) operații logice	9.5	1
(6) numărarea biților	28.5 sau 38	3 sau 4
Unități în virgula mobilă (64 biți)		
(7) adunare	57	6
(8) înmulțire	66.5	7
(9) aproximare reciprocă (RA)	133	14
Unități vectoriale (64 biți)		
(10) adunare întreagă	28.5	3
(11) deplasare	38	4
(12) operații logice	19	2
(13) numărarea biților	57	6
Operații cu memoria operativă (64 biți)		
Încărcare registre scalare	133	14
Încărcare registre vectoriale (64 elemente)	769.5	81

Toate unitățile funcționale sînt pipeline și pot accepta un nou set de argumente la fiecare perioadă de ceas. În lista de mai sus, numărul perioadelor este lungimea unității pipeline în ns sau perioade de ceas. La ultimele unități este variabila l utilizată în § 3.1. În cazul instrucțiunilor scalare, este intervalul de timp scurs de la lansarea în execuție a instrucțiunii pînă la obținerea rezultatului (ready) în registrul scalar, unde este disponibil pentru alte instrucțiuni. În cazul unei instrucțiuni vectoriale este necesară o perioadă de ceas suplimentară pentru a transfera fiecare operand de la un registru vectorial la o unitate funcțională, iar apoi o nouă perioadă pentru a transfera fiecare rezultat din pipeline în registrul vectorial. Deci, timpul de execuție a unei instrucțiuni vectoriale cu n elemente este :

$$t = (l + 2 + n)\tau \qquad (2.1a)$$

Comparînd ecuația (2.1a) cu formula generală (1.6) vedem că timpul de inițializare și lungimea performanței jumătate pentru operații vectoriale registru-la-registru sînt :

$$s = 3, \quad n_{1/2} = s + l - 1 = l + 2 \qquad (2.1b)$$

Unitățile de adresare calculează adresele, indecșii prin operații cu întregi mai mici decît aproximativ 16 milioane în complement față de 2. Operanzii și rezultatele se obțin, respectiv depun, în registrele A. Unitățile

Smith (1984) analizează modalități de acces la memorie mai realiste și ajung la concluzia că performanța se degradează în mod obișnuit cu 2,5 pînă la 7% în medie datorită conflictelor de acces la memorie, iar în situațiile cele mai defavorabile cu 20 pînă la 33%.

Dispozitivul SSD este un mediu de memorie MOS cu capacitatea de la 64 la 1024 MB (pînă la 128 Mw organizată în 128 blocuri), care poate fi folosit în locul discului magnetic, pentru memorarea datelor specifice unor probleme foarte complexe, sau de către sistemul de operare. De aceea, este denumit disc solid state și funcționează ca un disc cu timpul de acces mai mic de 50 microsecunde. Poate fi conectat la un X-MP/2 prin unul sau două canale de 120 MB/s, iar la un X-MP/2 cu un astfel de canal. X-MP/1 poate fi conectat cu un canal de 100 MB/s.

Calculatoare front-end, benzile magnetice, unitățile de disc, sînt întotdeauna conectate la CRAY X-MP prin subsistemul de I/E (IOS). Primul a fost introdus în 1979 pentru a crește posibilitățile de I/E ale calculatorului CRAY-1 inițial la mediile de memorare magnetică (benzi și discuri). Versiunea îmbunătățită produsă în 1981 este o parte integrantă a sistemelor CRAY X-MP instalate și constă din 2 pînă la 4 procesoare de I/E pe 16 biți (IOP) și un bufer de memorie de 8 Mw. Primul IOP gestionează comunicațiile cu pînă la 3 procesoare front-end, iar al doilea IOP gestionează comunicațiile cu pînă la 16 unități de disc și memoria comună a calculatorului X-MP via bufer. Al treilea și al patrulea IOP sînt gestionate și fiecare poate fi folosit pentru a adăuga încă 16 unități de disc. Unitatea de disc DD-49 produsă în 1984 are o capacitate de 1200 MB și o rată de transfer de 10 MB/s. De asemenea, există o cale opțională directă între IOS și SSD cu o rată de transfer de 40 MB/s. Toate unitățile centrale ale calculatorului CRAY X-MP partajează o singură secțiune de I/E care comunică cu IOS și SSD. Un canal de 1250 MB/s asigură transferul de date cu SSD, iar două canale de 100 MB/s realizează transferurile pentru IOS. 4 canale de 6 MB/s transferă cererile de I/E de la CPU și alte calculatoare la IOP.

Fiecare CPU are 4 bufere pentru instrucțiuni cu capacitatea de 128 grupuri de 16 biți fiecare. La X-MP/2 toate cele 8 porturi la memorie sînt folosite pentru extragerea instrucțiunilor operație care are loc cu o frecvență de 8 cuvinte (32 grupuri) într-o perioadă de ceas. Din acest motiv toate celelalte referințe la memorie ale celor două procesoare sînt suspendate în timpul unei extrageri de instrucțiuni de către oricare din procesoare. Un pachet de 16 cuvinte de 64 biți ce definește starea unui job utilizator poate fi transferat în 380 ns.

Secțiunea de intercomunicații a lui CRAY X-MP conține trei grupe de registre comune (5 grupe la X-MP/48), care pot fi accesate de către toate CPU pentru comunicații și sincronizare. De asemenea, există un ceas comun care permite evaluarea timpului de execuție al programului. Impulsurile de ceas furnizate tuturor unităților centrale sînt sincronizate. Registrele comune sînt 8 registre SB pe 24 biți, 8 registre ST pe 64 biți și 32 registre pe 1 bit pentru sincronizare sau semaforizare (SM). Anumite instrucțiuni realizează transferul conținutului lor în registrele A și S.

Cele 13 unități funcționale preiau date și depun date numai în registrele A, S, V, și în cele de mascare. Unitățile funcționale, care pot lucra

Aceasta poate fi recunoscută ca iterația Newton pentru inversul lui S2, deoarece este aceeași problemă cu cea a găsirii zeroului unei funcții

$$f(x) = S2 - x^{-1} \tag{2.3a}$$

care are derivata

$$f'(x) = x^{-2} \tag{2.3b}$$

Iterația Newton se definește cu

$$x^{(n+1)} = x^{(n)} - f(x^{(n)})/f'(x^{(n)}) \tag{2.3c}$$

deci

$$x^{(n+1)} = (2 - S2 * x^{(n)})\, x^{(n)} \tag{2.3d}$$

unde n este numărul iterației. În exemplul anterior, instrucțiunea realizează o aproximare a inversului lui S2 folosind unitatea de aproximare a inversului. Inversul este memorat de S3 și corespunde lui x din ecuațiile (2.3c, d). Instrucțiunea (2.2b) calculează expresia din paranteze din ecuația (2.3d) și este executată de unitatea de înmulțire în virgulă mobilă. Instrucțiunea de înmulțire (2.2c) multiplică aproximarea inițială a lui (S2) cu S1, iar (2.2d) aplică corecția specificată de ecuația (2.3d). După o iterație rezultatul se obține corect pe 47 biți, corespunzător preciziei mantisei pe 48 biți. Motivul multiplicării înaintea corecției este că instrucțiunile (2.2b) și (2.2c), deși folosesc aceeași unitate, pot fi lansate în execuție la perioade de ceas succesive; în timp ce instrucțiunea (2.2d), dacă ar fi executată înainte ar trebui să aștepte terminarea instrucțiunii (2.2b), deoarece are nevoie de valoarea lui S4. Împărțirea se încheie în 29 perioade de ceas, corespunzător unei viteze de 2,8 Mflop/s. Dacă instrucțiunile (2.2) au fost înlocuite de instrucțiuni vectoriale și scrise în ordinea (2a, 2c, 2b, 2d), instrucțiunile (2.2a) și (2.2c) se vor înlănțui împreună (vezi 3 paragrafe mai departe). Împărțirea a 2 vectori, element cu element, se va putea executa cu o viteză medie de 27 Mflop/s.

Instrucțiunile calculatorului CRAY X-MP sînt codificate fie cu un grup de (16 biți), fie cu 2 grupuri (32 biți). Înaintea execuției, instrucțiunile se află în 4 bufere pentru instrucțiuni, fiecare cu capacitatea de 128 instrucțiuni de un grup sau combinații echivalente din instrucțiuni de lungime diferită. Buferele sînt încărcate ciclic din memoria operativă. Cînd o instrucțiune necesară nu se află în bufer, se încarcă următorul bufer din ciclu cu cîte un cuvînt pe 64 biți (4 grupe) din fiecare bloc de memorie, în paralel. Există un numărător de program pe 22 biți (P) ce conține adresa următoarei instrucțiuni care urmează a fi executată, grupul următoarei instrucțiuni pe 16 biți (NIP) și un registru pe 16 biți al grupului curent (CP) pentru instrucțiunea ce așteaptă să fie lansată în execuție, ca și un registru pe 16 biți care păstrează grupul inferior al unei instrucțiuni cu 2 grupuri.

O instrucțiune poate fi lansată în execuție (trimisă unității funcționale pentru a fi executată), dacă unitatea nu este ocupată cu o operație și dacă registrele necesare pentru operanzi și rezultat nu sînt rezervate de alte instrucțiuni în curs de execuție. Instrucțiuni diferite pot rezerva re-

gistre, în modul prezentat de manual CRAY 1976). În linii mari, o instrucţiune vectorială, rezervă registrul ce urmează să primească rezultatul ca şi registrele cu operanzi pentru toată durata operaţiei. Dacă o instrucţiune vectorială foloseşte un registru scalar, el nu se rezervă, deoarece unitatea funcţională păstrează o copie a lui. Deci, în perioada de ceas ce urmează lansării în execuţie a instrucţiunii, conţinutul lui poate fi modificat. Similar, valoarea registrului ce păstrează lungimea vectorului, VL, poate fi modificată imediat după trecerea la execuţia instrucţiunii. De aceea, instrucţiuni cu vectori de lungimi diferite se pot executa concurent. În cazul instrucţiunilor scalare, instrucţiunea rezervă numai registrul pentru rezultat, pentru a preveni citirea lui de alte instrucţiuni, înainte ca valoarea lui să fie actualizată.

O caracteristică importantă a arhitecturii calculatorului CRAY X-MP este abilitatea sa de a *înlănţui* o serie de operaţii vectoriale, astfel încît să poată opera împreună ca un pipeline continuu (Johnson 1978). Diferenţa între operaţiile neînlănţuite şi cele înlănţuite este prezentată în fig. 2.6. Diagrama din zona superioară ilustrează evaluarea timpului de execuţie dacă nu se înlănţuie operaţiile.

Timpul pentru o operaţie neînlănţuită formată dintr-o secvenţă de m instrucţiuni vectoriale este

$$t = \sum_{i=1}^{m} \tau[s_i + l_i + (n-1)] \tag{2.4a}$$

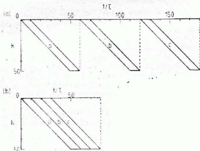


Fig. 2.6 Diagrama de timp (a) pentru trei operaţii vectoriale neînlănţuite, a, b, şi c şi (b) pentru aceleaşi operaţii dacă pot fi înlănţuite. În acest exemplu, $s + 1 = 10$ şi $nu = 50$. Secvenţa de prelucrare în timp a elementului k se obţine cu o linie orizontală trasată k unităţi sub axa timpului. Un element parcurge unitatea pipeline pentru prelucrare cînd aceată linie traversează trapezul corespunzător operaţiei respective.

Exprimat ca timp o operaţie vectorială devine

$$\frac{t}{m} = \frac{1}{\bar{r}_\infty}(\bar{n}_{1/2} + n) \tag{2.4b}$$

unde performanţa asimptotică medie este

$$\bar{r}_\infty = \tau^{-1} \tag{2.4c}$$

iar lungimea performanţei jumătate medie este

$$\bar{n}_{1/2} = \frac{1}{m}\sum_{i=1}^{m}(s_i + l_i - 1) = (s + l - 1) \tag{2.4d}$$

În cazul a m operaţii vectoriale înlănţuite avem

$$t = \tau\left(\sum_{i=1}^{m}(s_i + l_i) + (n - 1)\right) \tag{2.5a}$$

Se aplică ecuaţia (2.4b) dar cu

$$\bar{r}_\infty = m\tau^{-1} \tag{2.5b}$$

$$\bar{n}_{1/2} = \sum_{i=1}^{m}(s_i + l_i) - 1 = m(s + l) - 1 \tag{2.5c}$$

Rezultatele finale din ecuaţiile (2.4d) şi (2.5c) se obţin dacă toate unităţile pipe au acelaşi timp de iniţializare şi număr de etaje, aşa cum este de obicei. Din cele de mai sus vedem că o secvenţă de operaţii vectoriale neînlănţuite se comportă la fel ca o singură operaţie. Totuşi, în cazul înlănţuirii a m operaţii vectoriale ,atît performanţa asimptotică, cît şi lungimea performanţei jumătate cresc de m ori. Acest efect a fost asociat în cap. 1 cu multiplicarea procesoarelor. El intervine în cazul înlănţuirii deoarece cele m pipeline lucrează concurent, ceea ce în mod similar creşte r_∞ şi cantitatea de paralelism, $n_{1/2}$. În cazul operaţiilor neînlănţuite unităţile pipeline lucrează secvenţial şi de aceea rata medie de calcul sau paralelismul nu au cum să crească.

Arhitectura calculatorului CRAY X-MP poate fi prezentată pe scurt cu notaţia ASN din §1.2.4. Dacă se ignoră operaţiile de I/E şi detaliile de conectare a registrelor, se obţine:

$$C(\text{CRAY X-MP/nm}) = nC1(\text{CPU}) \times (8m)M1^{38}_{128K,64}(\text{common})\,;$$

$$C1(\text{CPU}) = Iv^{9.5}_{16.32}[14Ep - M2 - \{\langle \tfrac{9.5}{64}, \langle \tfrac{9.5}{64}, \tfrac{9.5}{64}\rangle\rangle\}]_r\,;$$

$$M2(\text{registers}) = \{8M_{64,64}(\text{vector}),\ 72M_{1,64}(\text{scalar}),$$
$$72M_{1,24}(\text{address})\}\,;$$

$$14Ep = \{3Fp_{64},\ 5Bp_{64},\ 4Bp_{64},\ 2Bp_{24}\}\,;$$

$3Fp_{64}(\text{floating-point}) = \{Fp_{64}(+),\ Fp_{64}(*),\ Fp_{6}^{4}(\div)\}$;

$5Bp_{64}(\text{vector}) = \{Bp_{64}(\text{integer}+),\ Bp_{64}(\text{shift}),\ 2Bp_{64}(\text{logical}),$

$Bp_{64}(\text{pop. count})\}$;

$4Bp_{64}(\text{scalar}) = \{Bp_{64}(\text{integer}+),\ Bp_{64}(\text{shift}),\ Bp_{64}(\text{logical}),$

$Bp_{64}(\text{pop. count})\}$;

$2Bp_{24}(\text{address}) = \{Bp_{24}(\text{integer}+),\ Bp_{24}(\text{integer}\ *)\}$

2.2.3 Tehnologia

Memoria operativă de 8 Mw a calculatorului CRAY X-MP/48 se compune din circuite VLSI de 64 Kb bipolare sau MOS statice cu un timp de acces de 38 ns sau 4 perioade de ceas. Spre deosebire de CRAY-1 care folosea circuite cu 4/5 porţi NAND, circuitele calculatoarelor CRAY X-MP, registrele A şi S sînt construite din circuite cu masive (array) de 16 porţi, definite printr-un timp de propagare de 300—400 ps. Conexiunile între module se realizează ca şi la CRAY-1 cu perechi răsucite de fire.

2.2.4. Setul de instrucţiuni

CRAY X-MP are aproximativ 128 instrucţiuni (cod de operaţie pe 7 biţi la majoritatea instrucţiunilor). Majoritatea instrucţiunilor conţin 3 adrese, specificînd sursa a doi operanzi şi destinaţia rezultatului. Deoarece toate unităţile funcţionale lucrează registru-la-registru, sînt necesari numai 3 biţi pentru definirea sursei sau destinaţiei, de unde situaţia că 3 adrese plus codul operaţiei pot fi codificate cu un grup de 16 biţi. Toate instrucţiunile, mai puţin 18, sînt cu un singur grup. Instrucţiuni care folosesc adrese de memorie (22 biţi) pentru transferuri de blocuri sau salt folosesc două grupuri şi sînt pe 32 biţi. Instrucţiunile care conţin constante pe 22 biţi ce urmează a fi transferate în registrele A sau B sînt, de asemenea, pe 32 biţi.

Instrucţiunile execută operaţii logice, de deplasare, de salt, de calcul aritmetic cu întregi sau numere reale în virgulă mobilă, în cele 13 unităţi funcţionale. Iată, în continuare, cîteva exemple în notaţii CAL (limbajul de asamblare). Formatul instrucţiunii este :

||g|h|i|j|k|| m||op codul este g sau (g/h)
||4|3|3|3|3|| 16 biţi|| (2.6)
||grup 1 ||grup 2||

Read/Write :

Bjk, Ai ,A0	Read (Ai) words to B register jk from (A0)
A0 Tjk, Ai	Store (Ai) words from T register jk to (A0)
Ai exp, Ah	Read from ((Ah)+jkm) to Ai. (A0=0, jkm=exp)
exp, Ah Si	Store (Si) to ((Ah)+jkm). (A0=0, jkm=exp)

Vi, A0, Ak	Read (VL) words to Vi from (A0) incremented by (Ak)
A0, Ak Vj	Store (VL) words from Vj to (A0) incremented by (Ak)
Ai Sj	Transmit (Sj) to Ai
Bjk Ai	Transmit (Ai) to Bjk
Si Tjk	Transmit (Tjk) to Si
SMjk 1	Set jth semaphore register
Ai SBj	Read jth shared B register

Control

J Bjk	Branch to (Bjk)
R exp	Return branch to ijkm (=exp); set BOO to instruction counter
JSZ exp	Branch to *ijkm* (=exp) if (SO)=0
JAP exp	Branch to *ijkm* (=exp) if (AO) positive

Logic:

Si Sj & Sk	Logical product of (Sj) and (Wk) to Si
Vi Sj ! Vk	Logical sum of (Sj) and (Vk) to Vi
SO Si < exp	Shift (Si) left jk(=exp)places to SO
Si Si, Sj < Ak	Shift register pair (SiSj) left (Ak) places to Si

Arithmetic:

Si exp	Transmit jkm (=exp) to Si
Ai Aj+Ak	Integer sum of (Aj) and (Ak) to Ai
Si Sj*Sk	Integer product of (Sj) and (Sk) to Si
Vi Sj*IVk	Floating product of (Sj) and (Vk) to Vi
Si /HSj	Floating reciprocal approximation of (Sj) to Si
Vi Vj*IVk	2.0-product of (Vj) and (Vk) to Vi; reciprocal iteration

Miscellaneous:

VL Ak	Transmit (*Ak*) to VL
Si VM	Transmit (VM) to Si
Si RT	Transmit (real-time clock) to Si
Ai PSj	Population count of (Sj) to Ai
Ai ZSj	Leading zero count of (Sj) to Ai

Parantezele asociate cu un număr de registre semnifică conţinutul acelui registru, iar exp. poate fi înlocuit cu o expresie aritmetică simplă, a cărei valoare este introdusă în instrucţiune:

Două instrucţiuni ale calculatorului CRAY X-MP merită o atenţie specială. Acestea sînt:

Set Mask — cei 64 biţi ai registrului vectorial mască (VM) corespund biunivoc celor 64 elemente ale unui registru vectorial. Dacă elementele satisfac o condiţie, bitul corespunzător din VM devine unu, altfel este

zero. Condiţiile sînt nul, nenul, pozitiv sau nul, negativ. Astfel:

VM V5, Z face 1 biţii VM unde elementele V5 sînt nule;

VM V7, P face 1 biţii VM unde elementele V7 sînt pozitive sau nule.

Vector Merge — conţinuturile a două registre vectoriale Vj şi Vk sînt combinate într-un singur vector rezultat Vi funcţie de registrul VM. Dacă bitul din poziţia 1 al registrului VM este 1 elementul 1 al lui Vj devine elementul 1 al registrului rezultat, altfel elementul 1 al registrului Vk este elementul 1 al rezultatului. Vj poate fi şi un registru scalar. Valoarea registrului de lungime al vectorului determină numărul elementelor care sînt combinate.

Astfel:

Vi Vj ! Vk & VM combină Vj cu Vk pentru a produce Vi funcţii de VM

V7 S2 ! V6 & VM combină S2 şi V6 pentru a produce V7 funcţii de VM.

Scopul instrucţiunilor de mascare şi combinare este de a permite evaluarea condiţionată cu instrucţiuni vectoriale. Să considerăm evaluarea

```
  DO 3 I=1,N
  IF (C(I)) 2, 1, 1
1 A(I)=EXPR 1
  GO TO 3                                          (2.7)
2 A(I)=EXPR 2
3 CONTINUE
```

unde A(I) ia valoarea expresiei EXPR 1 dacă C(I) este nul sau pozitiv, sau valoarea expresiei EXPR 2 dacă C(1) este negativ. Dacă presupunem că A(I) şi C(I) sînt memoraţi în elementul (I—1) al registrelor vectoriale V3, respectiv V4, iar VL conţine valoarea lui N (presupusă pentru simplitate ≤ 64), atunci codul de mai sus poate fi implementat cu următoarele instrucţiuni vectoriale.

(1) evaluează expresia EXPR 1 pentru toate elementele şi plasează-le în V1;

(2) evaluează expresia EXPR 2 pentru toate elementele şi plasează-le în V2;

(3) VM V4, P fă biţii VM 1 unde elementele $V4 \geq 0$;

(4) V3 V1 ! V2 & VM combină V1 şi V2 conform VM.

Observăm că în cadrul acestei metode expresiile EXPR 1 şi EXPR 2 trebuie evaluate pentru toate elementele, chiar dacă în vectorul final V3 vor fi folosite numai N din cele 2N rezultate. Evident, este o risipă de operaţii aritmetice, dar care permite folosirea instrucţiunilor vectoriale. Alternativ, codul FORTRAN (2.7) trebuie examinat element-cu-element, iar expresia EXPR 1 şi EXPR 2, evaluate aşa cum se arată, dar nu în acelaşi timp. În acest caz nu se execută operaţii aritmetice care nu sînt necesare; totuşi, operaţiile aritmetice scalare folosite sînt mai lente decît cele vectoriale ale primei metode. Evident, va exista o anumită lungime a vectorului de la care metoda de combinare vectorială este mai rapidă decît evaluarea cu instrucţiuni scalare.

Două situații pun probleme deosebite calculatoarelor paralele. Acestea sînt operațiile de dispersie (scatter) și grupare (gather), care pot fi definite cu următorul cod FORTRAN :

(1) *Scatter*

```
      DO 1 I=1, N
    1 X(INDEX(I))=Y(I)                                   (2.8a)
```

(2) *Gather*

```
      DO 1 I=1, N
    1 Y(I)=X(INDEX(I))                                   (2.8b)
```

În fiecare caz, masivul întreg INDEX conține un set de indici care pot specifica adrese care sînt distribuite arbitrar în memoria operativă a calculatorului. O operație de dispersie distribuie un set ordonat de elemente Y(I) în memorie, corespunzător șablonului de adrese al masivului INDEX. Operația inversă, de grupare, colectează elementele distribuite X și le sortează în masivul Y. Aceste operații sînt folosite în problemele de sortare; în reordonare ca, de exemplu, la transformata Fourier rapidă și în asignarea sarcinii (operație de distribuire) și interpolare a cîmpului (operație de grupare) în codul de simulare pentru un model de structuri de particule (vezi Hockney și Eastwood 1981).

Modelele CRAY X-MP anterioare celui cu 4 CPU anunțat în 1984, ca și calculatoarele anterioare CRAY-1 și CRAY-1S nu aveau hardware sau instrucțiuni pentru implementarea operațiilor de dispersie și grupare. Bucle ca (2.8) trebuie executate de instrucțiuni scalare și au o performanță dezamăgitoare de aproximativ 2.5 Mop/s (Hockney și Jesshope 1981). Calculatorul CRAY X-MP/4 posedă instrucțiuni de dispersie și grupare ca și instrucțiuni de comprimare a indecșilor care fac ca aceste operații să se execute la viteze vectoriale mult mai mari:

,A0, Vi Vj vector scatter Vj cu folosirea indicilor
Vi, A0, Vj vector gather în Vi folosind indicii Vj
Vi, VM Vj, Z comprimarea indicilor de Vj zero, obținerea indicilor elementelor zero ale lui Vj ca un vector comprimat în Vi și se face VM=1 în pozițiile binare corespunzătoare.

Instrucțiunea de grupare se execută cu unul din cele două porturi de citire, iar instrucțiunea de dispersie în portul de scriere. Instrucțiunea de comprimare a indicilor se execută în unitatea logică vectorială.

Folosind aceste instrucțiuni, bucla (2.8a) poate fi implementată cu

V0 ,A0, 1 încărcarea vectorului Y(I) in V0
V1 ,A0, 1 încărcarea vectorului INDEX(I) în V1 (2.8c)
,A0, V1 V0 vector scatter la X(INDEX(I)),

iar bucla pentru operația de grupare cu

V0 ,A0, 1 încărcarea vectorului INDEX(I) în V0
V1 ,A0, V0 gather X(INDEX(I)) în V1 (2.8d)
,A0,1 V1 salvarea vectorului Y în memoria comună

Metoda de vectorizare condiționată a expresiilor explicată în asociere cu instrucțiunea de combinare vectorială este ineficientă, deoarece operațiile aritmetice ale ambelor alternative ale instrucțiunii IF din (2.7)

trebuie executate ; chiar dacă este necesară o singură alternativă — rezultatele nedorite sînt abandonate în cursul operației de combinare. Dacă numărul elementelor care urmează să fie modificate prin execuția instrucțiunii IF reprezintă un procent mic din numărul total, o strategie mai bună este a culege indicii elementelor vectorului care satisfac condiția cu instrucțiunea de comprimare a indicilor și apoi de a grupa elementele într-un vector comprimat. Apoi se execută operațiile aritmetice numai cu vectorul comprimat care este mult mai scurt decît originalul. Rezultatele pot fi apoi dispersate în locațiile lor originale. Ca un exemplu banal să considerăm bucla :

```
      DO 1 I=1, N
    1 IF(X(I) .NE. 0)Y(I)=V(I)+X(I)                    (2.8e)
```

unde presupunem că numărul elementelor nenule ale lui X este mic. Acest cod se poate executa cel mai eficient cu

```
V0 ,A0, 1 încărcare vector X(I).
V1, VM V0, N pune indicii valorilor nenule ale lui
V5 V1  X(I) în V1, V5 și V6 ; și face 1
V6 V1 elementele corespunzătoare ale lui VM.
S1 VM pune vectorul mască în S1
A1 PS1 numără biții 1 din S1
VL A1 introduce lungimea vectorului comprimat în VL    (2.8f)
V2 ,A0, V1 vector gather X(I)
V3 ,A0, V5 vector gather Y(I)
V4 V2+FV3 Y(I)=Y(I)+X(I) cu vectorii comprimați
,A0, V6 V4 vector scatter a lui Y(I) în locațiile inițiale din memorie
```

Ca și in cazul anterior, textul trebuie executat cu toate cele N elemente ale vectorilor, deși operațiile aritmetice (care pot fi substanțiale într-un exemplu real) se execută numai cu vectorii comprimați. Dacă modul de acces la memorie evită conflictele, cele 4 instrucțiuni vectoriale finale din (2.8f) se înlănțuie și se pot executa la viteza vectorială maximă, de un element la fiecare perioadă de ceas.

2.2.5. Software

Principala trăsătură specifică nouă a soft-ului calculatorului CRAY X-MP în comparație cu CRAY-1 o reprezintă rutinele de sistem care fac posibilă cooperarea mai multor CPU la rezolvarea unei singure probleme. Această posibilitate, denumită multitasking la CRAY X-MP, asigură sincronizarea multiplelor CPU. La alte calculatoare asigură condițiile pentru așa numita programare paralelă, MIMD sau cu mai multe instrucțiuni (sau multiprocesare). CRAY rezervă termenul de multi-tasking pentru programarea paralelă la nivelul subrutinelor, unde unitatea de lucru executată de unitatea centrală este subrutina, preferabil mare. O astfel de unitate se cheamă task. Soft-ul oferă trei posibilități de bază : în primul rînd pentru inițierea și așteptarea unor taskuri (software TASKS sau

fork/join), în al doilea rînd protejarea secțiunilor critice de cod care trebuie executate numai de un CPU la un moment dat (software LOCKS) şi, în al treilea rînd sincronizarea secvențelor de evenimente care intervin în taskuri diferite (software EVENTS). Toate posibilitățile pot fi exploatate prin apeluri la subrutine FORTRAN din biblioteca multitasking standard.

În mod obişnuit un task se defineşte ca o subrutină şi este inițiat cu

CALL TSKSTART (*taskid, name, list*)

unde *taskid* este numele unui masiv întreg cu 3 elemente care identifică task-ul, *name* este numele subrutinei, iar *list* este o listă de argumente ale subrutinei. Un apel la TSKSTART defineşte un nou CPU logic care are o corespondență biunivocă cu taskul şi îl plasează într-o coadă de aşteptare pentru execuție. În acest mod, numărul de CPU nu este important, programele executîndu-se fără modificare pe modele cu 1—, 2— sau 4— CPU.

Un program multi-tasking va avea întotdeauna un program de control care va face unul sau mai multe CALL la TSKSTART la un punct în program unde se poate economisi timp prin atribuirea unor segmente independente de program diferitelor CPU. Un astfel de punct în program este denumit *bifurcație* (fork).

Posibilitatea asociată este

CALL TSKWAIT (*taskid*)

care determină programul de control să aştepte pînă ce identificarea task-ului *taskid* s-a încheiat. Dacă acest lucru s-a realizat pentru toate task-urile inițiale, s-a produs o *joncțiune* (join) în program (sau punct de sincronizare) care nu poate fi trecut pînă ce nu s-au încheiat toate fluxurile paralele de instrucțiuni din task-urile separate.

Adesea task-urile inițiate la un fork sînt independente şi nu au nevoie de comunicații cu alte task-uri în cursul execuției. Definirea task-urilor independente este metoda cea mai clară de programare care conduce la programele cele mai clare şi mai uşor de depanat. Aceasta nu este întotdeauna posibilă. Un exemplu de task-uri independente este calculul transformatelor Fourier rapide pe toate cele 32 linii ale unei structuri 32×32, în cursul rezolvării ecuațiilor diferențiale parțiale (vezi §5.6.2, ecuația (5.133). Jumătate din transformate pot fi date fiecărei CPU a unui X-MP/2, fără comunicații între ele. Este necesar, numai, ca toate transformatele să se încheie înaintea execuției următoarei etape din algoritm. Pentru aceasta se execută un CALL la TSKWAIT.

Prin contrast, task-urile separate create la un fork pot conține secțiuni de cod care pot da răspunsuri greşite dacă sînt executate simultan de mai mult de un CPU. Acestea sînt secțiuni critice de cod, un exemplu fiind actualizarea unei variabile comune. Dacă presupunem că se execută două sau mai multe taskuri, fiecare conținînd următorul cod

```
COMMON S
S=S+1
```

Se intenționează incrementarea variabilei S cu 1 cînd fiecare task atinge această zonă de cod. Dacă CPU2 citeşte S în timp ce CPU1 este în cursul actualizării lui, dar înaintea memorării noii valori, CPU2 va citi valoarea veche în loc de a citi pe cea nouă. Actualizarea efectuată de CPU1 se va pierde, iar numărul din S va fi incorect. Singurul mod de a asigura că

actualizarea este corectă este prin asigurarea terminării execuţiei instrucţiunii S=S+1, inclusiv memorarea în S, de către un singur task la un moment dat. Acesta este exemplul cel mai simplu al unei secţiuni critice de cod care trebuie executată numai de un CPU la un moment dat. Exemple mai complicate implică segmente substanţiale de cod, dar toate au caracteristica citirii şi scrierii unei variabile comune. Eşecul identificării şi protejării secţiunilor critice de cod este una din erorile de programare cele mai frecvente şi cele mai greu de detectat, deoarece uneori programul poate lucra corect, funcţie de anumiţi parametri ai calculatorului.

Soft-ul multi-tasking asigură protejarea (blocarea-locks) secţiunilor critice de cod, care sînt manipulate de următoarele subrutine

CALL LOCKASGN (*name*)
CALL LOCKON (*name*)
CALL LOCKOFF (*name*)
CALL LOCKREL (*name*)

unde *name* este o variabilă întreagă şi identificînd blocarea. Un lock poate avea numai două stări, fie *blocat* (descris uneori ca „*on*" sau „*set*") sau *neblocat* (descris ca „*off*" , „*reset*" sau „*cleared*"). LOCKASGN creează situaţia de blocare şi o face inactivă (off). LOCKON comută blocarea, dacă era inactivă, sau suspendă task-ul dacă era activă. În ultimul caz, task-ul rămîne suspendat pînă ce blocarea este anulată de alt task. În acest moment, toate task-urile în aşteptarea anulării blocării îşi vor continua execuţia. LOCKASGN comută blocarea inactivă în mod necondiţionat, iar LOCKREL elimină blocarea şi eliberează identificatorul atribuit plecării, care poate fi folosit ulterior pentru alte scopuri. Astfel, blocări cu acelaşi nume (folosind aceeaşi cheie) pot fi folosite pentru secţiunile critice de cod în task-uri diferite scriind

LOCKON (*name*)
<code : ex. S=S+1>
LOCKOFF (*name*)

dar este important să ne reamintim că mecanismele funcţionează ca şi cum există numai *o cheie* pentru fiecare *nume* de blocare. Astfel se asigură ca la un moment dat să se poată executa numai unul din segmentele blocate (sau protejate). În general, trebuie minimizate secţiunile de cod protejate, deoarece blocările forţează segmentele de cod să se execute secvenţial, iar CPU vor înceta să lucreze, aşteptînd cheia blocării, pierzîndu-se avantajul procesării paralele cu mai multe CPU.

Un alt motiv de comunicare între task-uri poate fi nevoia de a stabili o anumită secvenţă de evenimente între două task-uri care interacţionează. Soft-ul de multi-tasking asigură următoarele primitive de sincronizare:

CALL EVASGN (*name*)
CALL EVWAIT (*name*)
CALL EVPOST (*name*)
CALL EVCLEAR (*name*)
CALL EVREL (*name*)

unde *name* este o variabilă întreagă care se asociază cu apariţia unui eveniment. EVASGN defineşte existenţa unui eveniment denumit *nume* care are două stări posibile: *posted* (s-a întîmplat) sau *cleared* (nu s-a întîmplat încă). EVWAIT verifică starea evenimentului. Dacă nu s-a întîmplat încă, programul apelant este suspendat şi aşteaptă consumarea lui, în

cealaltă situaţie programul îşi continuă execuţia. EVCLEAR şterge evenimentul consumat şi, deoarece EVWAIT nu modifică starea evenimentului, este apelat imediat după EVWAIT. EVREL elimină evenimentul şi eliberează numele variabilei. Prin definiţii corecte ale evenimentelor şi apeluri la EVWAIT şi EVPOST, între două task-uri care interacţionează se poate forţa orice secvenţă de evenimente. Primitivele LOCKS şi EVENTS au facilităţi similare, şi în mod evident, se pot simula una pe alta. Este cel mai bine de folosit blocările pentru protejarea secţiunilor critice şi a evenimentelor pentru sincronizarea lor.

Aşa cum se subliniază în manualele CRAY, multi-taskingul se realizează cu un anumit cost. Apelurile subrutinelor TASKS LOCKS şi EVENTS consumă timp, existînd un overhead caracteristic oricărui program executat în regim de multi-tasking şi care nu este prezent în acelaşi program dacă s-ar executa ca un singur task pe un CPU. Formulat altfel, prin multi-tasking *nu* se reduce numărul ciclurilor CPU necesare unui job ci, de fapt se creşte prin apelurile la rutinele de multi-tasking. Ce se realizează este reducerea timpului consumat pentru un job prin partajarea ciclurilor CPU între mai multe CPU. În concluzie, multi-taskingul va fi util cînd mărimea task-urilor, cunoscută ca granularitatea paralelismului, este suficient de mare pentru ca overhead-ul să fie neglijabil. De aceea, este important să se reducă numărul apelurilor la biblioteca multi-tasking şi să se exploateze paralelismul programului la cel mai înalt nivel posibil (de exemplu, între instanţe diferite ale buclelor exterioare ale programului). Pentru a permite efectuarea unei analize cantitative, s-a măsurat overhead-ul asociat cu fiecare din cele trei metode de sincronizare, rezultatele urmînd să fie prezentate în secţiunea următoare.

2.2.6. Performanţa

În prima ediţie a cărţii *Parallel Computer* (Hockney şi Jesshope 1981) s-a discutat pe larg performanţa calculatorului CRAY-1 original. În continuare prezentăm măsurătorile efectuate pe un CRAY X-MP/22 cu două CPU pentru parametrii r_∞, $n_{1/2}$ şi $s_{1/2}$ (vezi §1.3.3 şi §1.3.6 şi Hockney 1 985)). Prezentăm întîi măsurătorile lui r_∞ şi $n_{1/2}$ pe un singur CPU şi, apoi, overhead-ul, $s_{1/2}$, de sincronizare a 2 CPU prin diverse metode. Fig. 2.7 arată rezultatul execuţiei echivalentului codului (1.5) pentru operaţii diadice şi două tipuri de operaţii triadice. Pentru fiecare lungime a vectorului, N, din programul (1.5), s-a repetat măsurătoarea de 100 ori, iar valoarea minimă s-a reprezentat în figura 2.7. Rezultatul se obţine în modul standard prin trasarea liniei drepte celei mai bune şi înregistrarea inversului pantei ca r_∞, iar intersecţia negativă cu axa n ca $n_{1/2}$. Tabelul 2.1 rezumă rezultatele.

Primele trei cazuri sînt măsurători pentru instrucţiuni vectoriale. Cazul diadic foloseşte un singur pipeline vectorial cu toţi vectorii în memoria principală şi poate fi comparat cu valorile $r_\infty = 22$ Mflop/s şi $n_{1/2}=18$ obţinute anterior pe CRAY-1 (Hockney şi Jesshope 1981). r_∞ este de trei ori mai mare datorită, în primul rînd, existenţei celor trei porturi la memorie pe care le are CRAY X-MP, în comparaţie cu unul la CRAY-1. Timpul de iniţializare în microsecunde, $t_0 = n_{1/2}/r_\infty$ nu s-a modificat semnificativ: complexitatea suplimentară a accesului la memorie de la XM—P fiind

compensată de reducerea perioadei ceasului. Totuși, importanța acestui overhead, măsurat de $n_{1/2}$, este de trei ori mai mare la X—MP, deoarece s-ar fi putut executa în acel interval de timp de trei ori mai multe operații aritmetice decît la CRAY—1. Rata maximă la care poate furniza rezultate o unitate pipeline în virgulă mobilă este de un rezultat la fiecare perioadă de ceas de 9,5 ns adică 105 Mflop/s. Valoarea măsurată $r_\infty = 70$ Mflop/s este mai mică datorită timpului necesar pentru încărcarea registrelor vectoriale păstrează 64 elemente, overhead-ul este adunat la fiecare 64 elemente, unde devine perceptibil (fig. 2.7).

Următoarele două cazuri sînt măsurători pentru operatii vectoriale și utilizarea simultană a unităților pipeline de înmulțire și adunare în virgulă mobilă. În cel mai rău caz ne putem aștepta la o dublare a valorii lui r_∞, care se realizează dacă un operand este un scalar, dar nu se atinge în toate cazurile vectoriale. Valorile lui $n_{1/2}$ nu se modifică; timpul de inițializare se înjumătățește deoarece două instrucțiuni împart un singur timp de inițializare de 0,8 microsecunde.

Am executat un program de test diadic cu instrucțiuni scalare și am obținut $r_\infty = 5$Mflop/s și $n_{1/2} = 4$. Timpul de inițializare t_0 rămîne de 0.8 microsecunde dar acum este de importanță neglijabilă deoarece viteza de execuție a operațiilor aritmetice este de 20 ori mai mică.

S-au folosit 4 metode de sincronizare a operării celor două CPU ale unui calculator CRAY X-MP pentru un singur job. Toate programele folosite sînt date în lucrarea lui Hockney (1985a). Ele constau în folosirea primitivelor TSKSTART și TSKWAIT,—metodă pe care o vom numi TASKS, a primitivelor LOCKON și LOCKOFF — metodă pe care o vom numi LOCKS, a primitivelor EVPOST și EVWAIT — metodă pe care o denumim EVENT și în sfîrșit, a metodei LOCKS simplificată scrisă în cod CAL. În toate cazurile programele au fost executate în modul stand-alone, iar evaluarea timpului de execuție s-a realizat cu funcția RTC (DUM), de ceas în timp real. Ne-am asigurat în acest mod că a doua unitate centrală fizică este atribuită celei de-a doua unități logice CPU și că măsurăm timpul de execuție a întregului job. În cazul metodei TASKS (fig. 2.8), după apelul ceasului de timp real la începutul măsurătorii (T1), instrucțiunea de apel a TSKSTART furnizează celei de-a doua CPU o copie a subrutinei DOALL, pe care aceasta începe să o execute. Primul CPU, care interpretează programul de control MULTI, va executa o altă copie a subrutinei DOALL, prin apelul CALL DOALL. Apelul lui TSKWAIT are rolul de a asigura că ambele CPU au încheiat partea lor din job înaintea apelului ceasului pentru înregistrarea sfîrșitului măsurătorii (T2). Parametrii lui DOALL sînt folosiți pentru a asigura că cele două CPU execută operații elementare diferite (fiecare N/2) din totalul de M. La această metodă overhead-ul necesar lansării unui nou task intervine la fiecare bifurcație (fork) dintr-un segment de program care este împărțit între 2 CPU și de aceea este inclus în măsurători.

Timpul măsurat respectă formula

$$t = 45 + 3{,}2\ s/400\ \mu s$$

care conduce la valorile lui r_∞ și $n_{1/2}$ din tabelul 2.2. Următoarea metodă de sincronizare mai ieftină este LOCKS. În acest caz observăm (tabelul 2.2) că $s_{1/2} = 4000$, este aproximativ 2/3 din valoarea găsită la metoda

```
      PROGRAM MULTI
      COMMON/GLOBAL/A(400), B(400), C(400)
      DIMENSION IDT(2)
      EXTERNAL DOALL
      DATA B/400*1.0/, C/400*1.0/

      NMAX = 400
      IDT(1) = 2

      T1 = 9.5E-9*RTC(DUM)
      T2 = 9.5E-9*RTC(DUM)
      T0 = T2-T1

      DO 20 N = 2, NMAX, 2
      T1 = 9.5E-9*RTC(DUM)
      NHALF = N/2,
      NH1 = NHALF + 1

      CALL TSKSTART (IDT,DOALL, NH1, N)
      CALL DOALL (1, NHALF)
      CALL TSKWAIT (IDT)

      T2 = 9.5E-9*RTC(DUM)

      T = T2-T1-T0
      WRITE (6, 100) N, T
   20 CONTINUE

  100 FORMAT (' N: ', I4, 4X, 'TIME IN SECONDS:' F16.12)
      STOP
      END

      SUBROUTINE DOALL (N1, N2)
      COMMON/GLOBAL/A(400), B(400), C(400)

      DO 10 I = N1, N2
   10    A(I) = B(I)*C(I)

      RETURN
      END
```

Fig. 2.8 Program pentru măsurarea lui r_∞ şi $s_{1/2}$ cînd un job este împărţit celor două unităţi centrale ale lui CRAY X—MP/22, prin metoda de sincronizare TASKS.

TASKS. Pe de altă parte, metoda EVENTS este de 2 ori mai puţin costisitoare decît metoda LOCKS, cu $s_{1/2} = 2000$. Pentru a afla overhead-ul cel mai mic, John Larson de la Cray Research Inc a programat în CAL o formă simplificată a metodei LOCKS. Overhead-ul se reduce de 10 ori la $s_{1/2} = 220$. O analiză a codului CAL arată că nu există timp pierdut şi că este improbabil ca la CRAY X-MP să se obţină o sincronizare cu mai puţin overhead. Trebuie spus că în cadrul codului CAL, un CPU aşteaptă ca celălalt să-şi încheie activitatea prin testarea continuă a unui registru de sincronizare. Astfel se evită situaţia ca unitatea centrală aflată în aşteptare să execute altă activitate în acest interval de timp, de aceea această metodă poate fi cu greu acceptată ca o metodă generală de sincronizare.

Tabelul 2.2 **Valori măsurate pentru r_∞ şi $s_{1/2}$ cînd operaţiile diadice cu operanzi în memorie sînt împărţite între două CPU ale unui CRAY-X-MP22. Overhead-ul se măsoară separat cu TASS, LOCKS, EVENTS şi cod CAL pentru sincronizare. Overhead-ul de sincronizare este $t_0 = s_{1/2}/r_\infty$ în microsecunde.**

Metoda	r_∞ (Mflop/s)	$s_{1/2}$ (flop)	t_0 (μs)	$\pi = t_0^{-1}$ (k/s)
TASK	130	5700	45	22
LOCKS	140	4000	28	36
EVENTS	140	2000	14	71
LOCKS simplificată cod CAL	110	220	2	500

Notă: rezultatele sînt rotunjite în mod deliberat la 2 cifre. Precizie mai mare ar sugera o acurateţe mai bună.

2.2.7. CRAY-2 şi CRAY-3

Fig. 2.9 este o imagine impresionantă a calculatorului CRAY-2 şi a rezervorului pentru agentul de răcire (CRAY 1985). Calculatorul aflat în primul plan, care cîntăreşte aproape 3 tone, conţine un procesor frontal, 4 procesoare secundare, o memorie comună de 256 Megacuvinte şi toate sursele de alimentare şi firele de legătură. Carcasa este un cabinet cilindric de 1,2 m înălţime şi 1,4 m în diametru. Comprimarea dimensiunilor este remarcabilă, de fapt un CRAY X-MP cu 4 CPU plus sistemul de I/E şi dispozitivul solid state care se află într-un singur container ocupă o treime sau un sfert din volumul actual. Puterea generată de 195 KW diferă puţin faţă de cea de la CRAY X-MP, dar pentru eliminarea ei se foloseşte o tehnologie complet nouă de răcire — *răcire prin scufundare în lichid*. Toate plăcile cu circuite şi sursele de alimentare sînt complet scufundate într-o baie de lichid fluorocarbon care este deplasat lent (aproximativ 1 inch pe secundă) şi trecut printr-un sistem de schimbare a căldurii. Răcirea este foarte eficientă deoarece agentul de răcire, care are o constantă dielectrică mare şi proprietăţi bune de izolare este în contact direct cu plăcile cu circuite şi capsulele propriu-zise. Fig. 2.10 este o imagine mai clară a circulaţiei lichidului de răcire peste plăci. Sistemul de răcire se află în apropierea sistemului; coloanele din planul secund al fig. 2.9 reprezintă rezervorul pentru cele 200 galoane de lichid. Dacă trebuie înlocuită o placă, trebuie pompat în rezervor tot lichidul de răcire. Această operaţie se execută în cîteva minute.

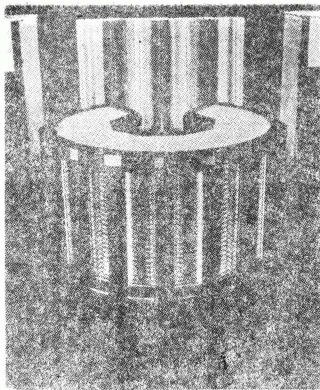


Fig. 2.9 O imagine de ansamblu a calculatorului CRAY-2, cu rezervorul cu lichidul de răcire în plan îndepărtat.

Fig. 2.10 Tehnologia de răcire prin imersiune totală a calculatorului CRAY-3. Toate plăcile cu circuite şi firele de conectare se află într-o baie de fluorocarbon. (Fotografie pusă la dispoziţie de Cray Research Inc.)

Perioada ceasului este la CRAY-2 de 4,1 ns, ceea ce impune folosirea unor fire de legătură scurte. Din acest motiv s-au proiectat modulele cu conexiuni după trei dimensiuni, fiecare fiind format din opt plăci cu circuite fixate rigid. Fiecare modul (vezi fig. 2.11) formează un masiv de $8 \times 8 \times 12$ capsule de circuite integrate. Modulul măsoară aproximativ $1 \times 4 \times 8$ inci și consumă între 300 și 500 W. Cele 320 module sînt montate în 14 coloane care formează un arc de 300°. Sînt 152 module pentru CPU și 128 module pentru memorie, cu aproximativ 240000 circuite, din care aproape 75000 sînt de memorie. Circuitele logice folosesc masive de 16 porți, ca la CRAY X-MP. Ca și la alte calculatoare CRAY, se folosesc perechi de fire răsucite cu lungime între 2 inci și 25 inci. Sînt folosite aproximativ 36000 conexiuni cu o lungime totală de aproximativ 6 mile.



Fig. 2.11 Un modul CRAY-2, format din 8 plăci cu circuite. (Fotografie furnizată de Cray Research Inc.)

Arhitectura generală a calculatorului CRAY-2 este prezentată în fig. 2.12 și poate fi descrisă ca formată din 4 procesoare secundare care accesează o memorie comună partajată, sub controlul unui procesor principal. Memoria comună de 256 M cuvinte de 64 biți este adresabilă direct de toate procesoarele (se folosesc adrese pe 32 biți). Este organizată ca 4 sferturi de 32 blocuri, realizîndu-se 128 blocuri cu 2 Megacuvinte pe bloc. Se folosește tehnologia MOS dinamică (256 kbiți pe capsulă), ceea ce înseamnă că timpul de acces (aproximativ 250 ns) este foarte mare în comparație cu memoria ECL principală de la CRAY X-MP (38 ns). În acest sens, este mai corect să se compare memoria comună cu memoria solid state de la X-MP. Fiecare bloc de memorie are o cale de date funcțională independentă la fiecare din cele 4 porturi bidirecționale de memorie, fiecare realizînd legătura cu un procesor secundar și un canal de comunicație. Lărgimea de bandă totală la memorie este de 1G cuvinte/s.

Accesul la memorie se face în fază (phased), adică fiecare procesor poate avea acces la un anumit sfert numai la fiecare 4 perioade, corespunzător fazei sale. Există 4 faze, cîte una pentru fiecare procesor secundar. Cele 32 blocuri de memorie dintr-un sfert partajează o cale de date la fiecare port de memorie comună; totuși, datorită modului de acces în fază, numai un bloc accesează calea în decursul a 4 perioade de ceas. Fiecare bloc are deci, funcțional, o cale independentă la fiecare din cele 4 porturi. Cei mai puțini semnificativi 2 biți ai adresei de memorie selectează sfertul, următorii 5 unul din cele 32 blocuri și cei 25 care rămîn specifică cuvîntul din bloc (bineînțeles, acum sînt necesari numai 21 biți pentru a adresa un bloc de 2 Mcuvinte). Astfel, elementele unui vector, memorate

continuu sînt răspîndite în primul rînd în sferturi, apoi în blocuri și, în sfîrșit, în cuvintele unui bloc.

Dacă referințele la elemente succesive ale unui vector au loc în fiecare perioadă de ceas, atunci un anumit sfert va fi referit la fiecare 4 perioade de ceas, iar un anumit bloc la fiecare 128 perioade de ceas (512 ns). Timpul de acces la memorie de 250 ns asigură evitarea conflictelor de acces la bloc în cazul ideal al referirii unui vector memorat continuu. Conflictele

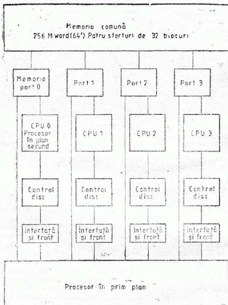


Fig. 2.12. Diagrama bloc a unui calculator CRAY-2 cu patru CPU.

vor apare dacă diferența între adresele elementelor succesive este 4 sau o putere mai mare a lui 2. În cazul cel mai defavorabil, cînd toate elementele sînt memorate în același bloc, rata de acces la memorie este de numai 1/64 din maximul atins pentru vectori continui. În consecință, performanța calculatorului CRAY-2 depinde în mod critic de modul de acces la memoria comună și poate varia în limite largi pentru job-uri diferite sau implementări diferite ale aceluiași program.

Procesorul principal supervizează procesoarele secundare, memoria comună şi controlerele de periferice prin 4 canale de comunicaţie de 4 Gbit/s. Fiecare canal este un inel pe 16 biţi ce conectează un procesor secundar la pînă la 9 controlere de disc, o interfaţă front-end, un port al memoriei comune şi procesorul principal. Inelul transferă un pachet de 16 biţi între staţii la fiecare perioadă de ceas. Procesorul principal este un calculator care lucrează cu întregi pe 32 biţi, şi are o memorie locală de date de 4 Kcuvinte (32 biţi) şi o memorie pentru instrucţiuni de 32 KB.

Arhitectura unui procesor secundar, prezentată în fig. 2.13, poate fi descrisă ca o arhitectură CRAY-1 cu registrele intermediare B şi T înlocuite de o memorie locală de 16 Kw. Există, ca şi la CRAY-1, o singură cale bidirecţională de date la memoria comună, 8 registre vectoriale a 64 cuvinte şi 8 registre scalare a 64 biţi. Cele 8 registre de adresă au cîte 32 biţi dar nu mai există o cale directă între memoria comună şi registrele de adresă. Şi unităţile funcţionale sînt aranjate într-un mod diferit, iar numărul lor a fost redus la 9. Nu există unitate de aproximare a inversului în virgulă mobilă separată. Această funcţie se execută acum în unitatea de înmulţire în virgulă mobilă, care execută şi funcţia nouă hardware de extragere a rădăcinii. Deplasarea vectorială, numărarea biţilor şi operaţiile cu numere întregi se execută în aceeaşi unitate vectorială; adunarea scalarilor întregi şi numărarea biţilor se execută în unitatea scalară întreagă. Exceptînd faptul că sînt unităţi pe 32 biţi, unităţile funcţionale de adrese sînt aceleaşi ca la CRAY-1. Memoria locală de 16 Kw are timpul de acces de 4 perioade de ceas şi are rolul de a memora temporar scalarii şi segmente de vectori în cursul calculelor. Viteza acestei memorii principale relative la unităţile funcţionale este aceeaşi cu a memoriei de la CRAY X-MP, care este, bineînţeles, mult mai mare (pînă la 8 Mw).

Fiecare procesor secundar are propriul ceas de timp real pe 64 biţi care avansează la fiecare perioadă de ceas şi este sincronizat iniţial cu ceasurile celorlalte procesoare secundare. Există un numărător pe 32 biţi pentru adresa programului, iar o bază de 32 biţi plus registre limită asigură protecţia domeniului de memorie comună. 8 semafoare de 1 bit şi un registru de stare pe 32 biţi controlează accesul la zonele de memorie comună şi sincronizarea procesoarelor secundare. 8 bufere pentru instrucţiuni păstrează 64 instrucţiuni de 16 biţi, iar o instrucţiune este lansată în execuţie la fiecare perioadă de ceas. Cele 128 coduri de instrucţiuni includ posibilităţi de dispersare/grupare, ca la CRAY X-MP/4; totuşi, posibilitatea înlănţuirii unei succesiuni de instrucţiuni vectoriale, care este o proprietate importantă la CRAY-1 şi X-MP, nu este disponibilă la CRAY-2.

Sistemul de operare al calculatorului CRAY-1 se bazează pe sistemul AT şi T Unix, care a devenit un standard industrial (Ritchie şi Thompson 1974, Bourne 1982). El posedă facilităţile Unix standard şi un compilator C. Un nou compilator FORTRAN optimizat (MFT) şi un asamblor CAL-2 însoţesc posibilitatea de conversie în cod asamblor de la CRAY-1 la CRAY-2. La Unix s-au introdus extensii pentru a creşte performanţa de I/E, posibilităţile de multiprocesare şi conectare în reţea.

Performanţa maximă a unui procesor secundar este de două rezultate în virgulă mobilă la fiecare perioadă de ceas, adică 500 Mflop/s pentru un calculator cu 4 CPU. Aceste valori s-ar obţine dacă argumentele şi

rezultatele s-ar afla în registre şi memorie locală (operaţie registru la registru) şi pot fi obţinute pentru probleme favorabile ca înmulţirea matricelor, pentru care s-a raportat 430 Mflop/s. Vitezele obţinute pentru job-uri FORTRAN medii, care ar utiliza memoria comună lentă, vor depinde în

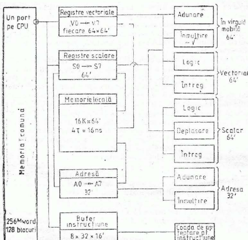


Fig. 2.13 Diagrama bloc a unui calculator CRAY-2 cu un CPU.

mare măsură de eficienţa cu care compilatorul buferizează transferurile şi de numărul mediu de operaţii aritmetice executate pentru o referinţă la memorie. Valorile obţinute pentru testul (r_∞, $n_{1/2}$), (vezi §1.3.3), pot fi mai elocvente privind performanţa unor job-uri FORTRAN medii care lucrează cu memoria comună.

Tabelul 2.3 prezintă rezultatele execuţiei testului (r_∞, $n_{1/2}$ pe CRAY-2, pentru o diversitate de programe simple, cu folosirea primului compilator CIVIC (1985) şi a unui compilator îmbunătăţit, CFT77 (1.3) care a apărut doi ani mai tîrziu. (La aceste teste, toate datele de intrare şi rezultatele se păstrează în memoria comună, memoria locală de 16 Kw nefiind folosită. Se execută puţine operaţii aritmetice pentru o referinţă la memorie ($f < 2$), astfel că performanţa este determinată în primul rînd de viteza memoriei comune, şi nu de cea a unităţilor pipeline aritmetice. De aceea, valorile măsurate ale lui r_∞ reprezintă o evaluare a situaţiei celei mai defavorabile, care poate apare ca urmare a execuţiei unui cod FORTRAN

care a nu fost optimizat pentru folosirea memoriei locale, prin directive către compilator. Dacă aceleași bucle s-ar executa cu date de intrare și rezultate memorate în registre vectoriale, performanța va tinde spre cea maximă a unei singure unități aritmetice pipeline ($\hat{r}_{\infty}$). Această coloană ar reprezenta, deci, cel mai bun caz posibil.

Primele 5 programe sînt vectorizate de compilator și se caracterizează prin creșterea performanței odată cu creșterea lui f, așa cum ne așteptăm la un calcul cu spațiu de memorie limitat. Următoarele două programe nu sînt vectorizate și performanța este specifică codului scalar. Ultimele două execută numai deplasări de date. Transpunerea are o performanță similară unei operații scalare. Operațiile de dispersare/grupare sînt implementate destul de eficient, ele executîndu-se la o rată apropiată de vitezele vectoriale. Rezultatele mai arată o degradare severă a performanței cînd se folosesc vectori ne-continui (cu o diferență între adresele elementelor succesive de 8). Raportul mare dintre rata teoretică maximă $\hat{r}_{\infty}$ și cea măsurată, r_{∞}, confirmă afirmația că performanța calculatorului CRAY-2 depinde în mod critic de modul de programare a problemelor, în particular de minimizarea referințelor și maximizarea utilizării memoriei locale pentru rezultatele intermediare.

Tabel 2.3 **Rezultate ale testului (r_{∞}, $n_{1/2}$) pentru o serie de programe executate de o unitate centrală a calculatorului CRAY-2, folosind cod FORTRAN și compilatorul CIVIC (1985). Valorile mai bune din paranteze sînt pentru CFT77 (1.3), Noiembrie 1987.**

Operația: instrucțiunea 10 în codul (1.5) Pas	Pas	$\hat{r}_{\infty}$ (Mflop/s)	r_{∞} (Mflop/s)	$n_{1/2}$
Diadă	1	244	32(56)	53(83)
A(I) = B(I) * C(I)	8	244	9(10)	10(0,5)
f = 1/3				
Triade vectoriale	1	488	54(65)	31(28)
A(I) = B(I) * C(I) + D(I)	8	488	14(17)	7(2,5)
f = 1/2				
Vectori cu 4 op.	1	488	73(100)	33(33)
A(I) = B(I) * C(I)+D(I)+E(I)				
÷ F(I)				
f = 2/3				
Înmulțirea matricelor 1		488	75(72)	75(79)
Produs intermediar (§ 5.3.2)				
f = 2				
Produs interior	1	488	73(119)	283(236)
S = S + B(I) * C(I)				
f = 1				
Recurență de prim ordin	1		3,1(12)	4,1(15)
A(I) = B(I) * A(I−1)+D(I)				
f = 1/2				
Atribuirea sarcinii	1	244	1,4(2,8)	2,6(10)
A(J(I)) = A(J(I)) + S				
f = 1/2				
Transpunere	1	—	2,1(3,4)	1,6(6,2)
Distribuire aleatoare; grupare —	—		19(29)	26(45)
A(J(I)) = B(I); A(I) = B(J(I))				

nițializare în microsecunde, $t_0 = n_{1/2}/r_\infty$ nu s-a modificat semnificativ: complexitatea suplimentară a accesului la memorie de la X-MP fiind compensată de reducerea perioadei ceasului. Totuși, importanța acestui overhead, măsurat de $n_{1/2}$ este de trei ori mai mare la X-MP, deoarece s-ar fi putut executa în acel interval de timp de trei ori mai multe operații aritmetice decît la CRAY-1. Rata maximă la care poate furniza rezultate o unitate pipeline în virgulă mobilă este de un rezultat la fiecare perioadă de ceas de 9.5 ns, adică 105 Mflop/s. Valoarea măsurată $r_\infty = 70$ Mflop/s este mai mică datorită timpului necesar pentru încărcarea registrelor vectoriale cu date din memoria principală. Deoarece registrele vectoriale păstrează 64 elemente, overhead-ul este adunat la fiecare 64 elemente, unde devine perceptibil (fig. 2.7).

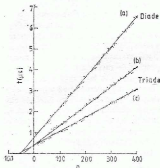


Fig. 2.7 Măsurători ale lui r_∞ și $n_{1/2}$ pe un CPU al unui calculator CRAY (X–MP/22. Timpul t, ca funcție de lungimea vectorului, n, pentru o singură operație vectorială. (a) Operații diadice A = = B × C. (b) Toate operațiile vectoriale triadice A = D × B + C)c. CYBER 205, triada, A = sB + + C. Toți vectorii sînt obținuți din și returnați în memoria comună. (Figură furnizată de North Holland, din Parallel Computing).

Următoarele două cazuri sînt măsurători pentru operații vectoriale triadice, care implică înlănțuirea a două instrucțiuni vectoriale și utilizarea simultană a unităților pipeline de înmulțire și adunare în virgulă mobilă. În cel mai bun caz ne putem aștepta la o dublare a valorii lui r_∞, care se realizează dacă un operand este un scalar, dar nu se atinge în toate cazurile vectoriale. Valoarea lui $n_{1/2}$ nu se modifică; timpul de inițializare se înjumătățește deoarece două instrucțiuni împart un singur timp de inițializare de 0.8 microsecunde.

Tabelul 2.1 Măsurători ale valorilor lui r_∞ și n în cazul unui CRAY X-MP cu un CIU care execută operații cu operanzi din memorie. În paranteze sînt valorile corespunzătoare lui CRAY-1. Timpul de inițializare este $t_0 = n_{1/2}/r_\infty$

Operație : instrucțiunea 10 din codul (1.5)	r_∞ (Mflop/s)	$n_{1/2}$ (flop)	t (μs)
Diada A(I) = B(I) × C(I)	70	53	0,75
(valori CRAY-1)	(22)	(18)	(0,82)
Triade vectoriale A(I)= D(I) × B(I) + C(I)	107	45	0,42
Triadă CYBER 205	148	60	0,40
Cod scalar A(I) = B(I) × C(I)	5	4	0,80

CRAY-3 este o implementare a calculatorului CRAY-2 în tehnologie cu arseniură de galiu (GaAs) care ar permite o perioadă a ceasului de 1 ns. Capsulele sînt produse de Gigabit Logic Inc., California (Alexander 1985) și Harris Microwave (Mc Crone 1985). Probabil CRAY-3 va avea mai multe procesoare, de la 8 la 16, și o memorie comună de 1 Gw. Este planificat pentru 1988/9.

2.3. CDC CYBER 205 și ETA[10]

CYBER 205 este produs de Control Data Corporation la Saint Paul, Minnesota, SUA. Mașina a fost anunțată în 1980, iar prima livrare la un utilizator s-a realizat la Oficiul meteorologic din Bracknell, Marea Britanie în 1981. Pînă în anul 1985 au fost instalate aproximativ 27 CYBER 205. CYBER 205 reprezintă punctul culminant al unui program lung de cercetare și dezvoltare, care a început cu proiectarea și livrarea calculatorului CDC STAR 100 în perioada 1965—1975 (vezi §1.1.3). Neil Lincoln, șeful echipei de proiectare prezintă tehnologia și compromisurile efectuate în cursul creerii acestui calculator (1982). Alte detalii se pot afla în CDC CYBER 200 Model 205 Computer System Hardware Reference Manual (CDC 1983). În 1983 s-a lansat CYBER 205, seria 600, la care s-a înlocuit memoria principală bipolară de 4 Mw a calculatorului inițial (denumit acum seria 400) cu o memorie MOS statică de 16 Mw. În continuare se descriu ambele mașini. Deoarece ele diferă numai prin tehnologia și încapsularea memoriei, vom descrie în continuare mașina seria 400 cu 4Mw și 2 unități pipeline și vom sublinia, cînd este cazul, diferențele față de seria 600. Arhitectura CYBER 205 este importantă deoarece este adoptată și de următoarea generație de supercalculatoare. Calculatorul ETA[10] care a apărut în 1986 poate fi definit ca 8 calculatoare CYBER 205 care lucrează cu o memorie comună mare. El este descris în §2.3.7.

2.3.1. Structura fizică

In fig. 2.14 și 2.15 apare o fotografie a calculatorului CYBER 205 și o diagramă bloc a diferitelor părți ale mașinii. Fotografia, luată din partea stîngă sus, este a unei mașini cu 2 unități pipeline. Memoria principală este formată din 4 sau 8 secțiuni. La seria 400 memoria se află în 2 sau 4 cabinete, fiecare conținînd 2 secțiuni a 1 milion cuvinte de 64 biți.
Secțiunea scalară care conține unitatea de prelucrare a instrucțiunilor formează partea centrală a mașinii. La un capăt se află memoria, conectată printr-o unitate de interfațare, iar la celălalt procesorul vectorial. Acesta conține una, două sau 4 unități aritmetice pipeline în virgulă mobilă, și secțiunea de I/E și reparare. Un calculator cu 4 Mw ocupă 7,7m×7 m. Răcirea calculatorului central cu 1 Mw de memorie se realizează cu 2 unități cu apă de 30t, iar energia este furnizată de un generator de 250 KVA. Se disipă aproximativ 118 KW. Mai sînt necesare un generator suplimentar de 80 KVA și un sistem de răcire de 90t pentru sistemul de memorie de 4 Mw. De asemenea, se asigură un generator de 250 KVA

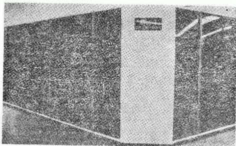


Fig. 2.14 Imagine a calculatorului CDC CYBER 205. (Fotografie pusă la dispoziţie de Control Data Corporation.)

Vectorial în virgulă mobilă (opţional, pînă 3 şi 4) secţia S

Procesor vectorial

Flux vectorial şi pentru şiruri secţia P

I/E şi iniţializare vectorial secţia N

Secţie scalară L

Procesor scalar

Vectorial în virgulă mobilă (pipe 1 şi 2) secţia R

Memorie 512K secţia H

Memorie 512K secţia A

Memorie 512K secţia G

Secţia K

Interfaţă memorie

Secţia J

Memorie 512K secţia B

Memorie 512K secţia F

Memorie 512K secţia E

Memorie 512K secţia D

Memorie 512K secţia C

Fig. 2.15 Diagrama bloc a diferitelor unităţi ale calculatorului CDC CYBER 205, seria 400. Seria 600 diferă numai la secţiunea de memorie, care este mai mică şi este dreptunghiulară în plan. (Fotografie furnizată de Control Data Corporation.)

pentru situații de urgență .CYBER 205 este proiectat să fie conectat la un sistem front-end, de obicei un CYBER 180, CDC 6000, IBM sau VAX.

Seria 600 diferă față de descrierea anterioară numai prin cabinetele pentru memorie care se situează la capătul secțiunilor J și K, formînd împreună un plan rectangular. Există tot 4 sau 8 secțiuni de memorie, care conține fiecare 0,5 , 1, 1,5 sau 2 Mw fiecare, realizînd configurațiile de 1, 2, 4, 8, 12 sau 16 Mw memorie.

2.3.2. Arhitectura

Unitățile componente și căile de date ale calculatorului CYBER 205 sînt prezentate în fig. 2.16. Memoria este formatădin 8 *secțiuni* (A la H) fiecare împărțită în 8 stive (denumite module de memorie la seria 600). Fiecare stivă (sau modul) este împărțită în 8 blocuri de memorie și are o cale independentă pe 32 biți la unitatea de interfațare a memoriei. Fiecare bloc conține 16 K jumătăți de cuvinte de 39 biți la seria 400 (32 biți pentru date plus 7 biți SECDED). La seria 600, un bloc poate conține 16K, 32K, 48K sau 64K jumătăți de cuvinte, funcție de numărul de socluri montate pe placă (vezi §2.3.3).

Memoria este organizată în perechi de secțiuni (A/H, B/G, C/F, D/E) fiecare avînd 16 stive și o cale de date de 512 biți la unitatea de interfațare. Această mărime a datei este cunoscută ca *supercuvînt* sau *sword.* Este echivalent cu 8 cuvinte de 64 biți sau 16 jumătăți de cuvinte a 32 biți și reprezintă unitatea de acces la memorie pentru vectori. Adresele succesive ale unui sword sînt memorate în stive diferite, oferind posibilitatea accesului în paralel prin extragerea unei jumătăți de cuvînt de la fiecare stivă din cele 16 ale unui sistem cu 2 secțiuni de memorie. Timpul de acces la memorie este de 80 ns. Dacă se execută referințe succesive, din fiecare secțiune dublă se poate accesa un nou sword la fiecare perioadă de ceas de 20 ns. Se obține, în acest caz o lărgime a benzii 400 Mw/s, pentru o secțiune dublă de memorie. Totuși unitatea de interfațare a memoriei nu folosește pe deplin această posibilitate (vezi în continuare).

Deși memoria poate fi adresată la nivel de bit, bait (8 biți), jumătate de cuvînt (32 biți) sau cuvînt (64 biți), accesul la memorie se realizează pe sword (512 biți) pentru vectori, în celelalte situații pe cuvînt sau jumătate de cuvînt. Unitatea de interfațare a memoriei organizează cererile la memorie în cursul fiecărui interval de 20 ns în sword, cuvinte sau jumătăți de cuvinte, apoi livrează sau asamblează datele prin căi de 128 biți, unităților scalare sau vectoriale. Comunicația cu restul calculatorului se realizează prin 3 căi pentru citire și două pentru scriere, unitatea de interfațare avînd un bufer cu capacitatea de 1 sword pentru fiecare cale (R1, R2, R3 pentru citire și W1, W2 pentru scriere). Unitatea de interfațare se conectează la secțiunile scalară, vectorială sau de I/E prin 10 căi de date de 128 biți, fiecare cu o rată de transfer maximă de 128 biți la fiecare perioadă de ceas, obținîndu-se o viteză de transfer maximă de 1000 Mw/s.

La o mașină cu 2 unități pipeline, unitatea de interfațare a memoriei operează așa cum s-a prezentat mai sus, cu o viteză maximă de citire de un sword sau 8 cuvinte pe o perioadă de ceas. Cum fiecare pipe necesită

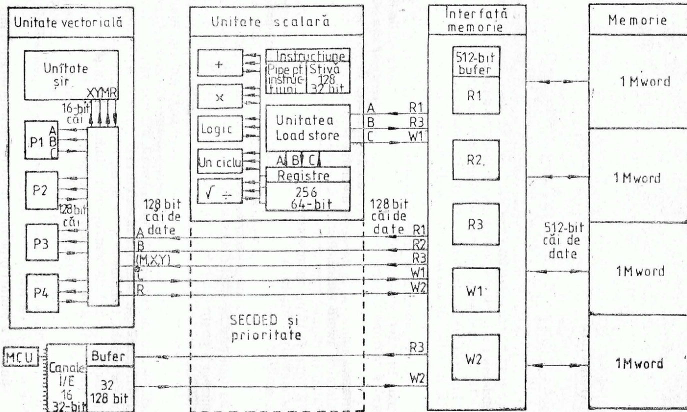


Fig. 2.16 Diagrama bloc care prezintă principalele unități și căi de date ale calculatorului CDC CYBER 205, seria 400. Seria 600 are memoria mărită la 16Mw.

două noi argumente de intrare la fiecare perioadă de ceas, capacitatea căilor de date și a interfeței tocmai satisfac aceste condiții ale unităților aritmetice pipeline. O mașină cu 4 unități are nevoie de un număr dublu de date într-o perioadă de ceas, de aceea buferele R1, R2 și W1 au capacitatea mărită la 1024 biți, iar căile de date la unitățile vectoriale sînt de 256 biți. Unitatea de interfațare execută referințe simultane la un dublu sword (1024 biți), prin accesarea simultană a unei jumătăți de cuvînt din 32 stive răspîndite în 4 secțiuni de memorie. Evident nu se utilizează întreaga capacitate de comunicație a memoriei, care poate furniza o jumătate de cuvînt simultan de la fiecare din cele 64 stive (8 stive în fiecare din cele 8 secțiuni de memorie).

Secțiunea scalară citește din buferele R1 și R3 și scrie în buferul W1. Toată căile de date la secțiunea vectorială traversează secțiunea scalară unde se realizează verificarea SECDED și determinarea priorităților cererilor la memorie. Includerea verificării SECDED extinde considerabil timpul mediu de funcționare între defectări ale sistemului. Secțiunea scalară conține unitatea pipeline de lansare în execuție a instrucțiunilor care au o cadență maximă de 1 instrucțiune la fiecare 20 ns (τ). Instrucțiunile cu 3 adrese sînt extrase dintr-o stivă care poate înregistra pînă la 128 de instrucțiuni pe 32 biți sau 64 instrucțiuni pe 64 biți, sau combinații cu lungimea totală echivalentă. Atît instrucțiunile vectoriale, cît și scalare sînt decodificate în unitatea pipeline de lansare în execuție, care trimite instrucțiunile vectorilor decodificate unității vectoriale pentru execuție. Trimiterea instrucțiunilor scalare decodificate unităților scalare aritmetice este controlată de un sistem de rezervare, principalele condiții fiind :

(1) *conflictul operandului sursă* — o instrucțiune care are nevoie de rezultatul furnizat de o instrucțiune anterioară trebuie să aștepte disponibilitatea acestuia :

(2) *conflictul operandului produs* — o instrucțiune al cărui rezultat referă aceiași registru ca și o instrucțiune lansată anterior în execuție trebuie să aștepte încheierea execuției acesteia.

16 registre pentru adresa rezultatului păstrează adresele registrelor necesare pentru operanzii rezultați în urma execuției unor instrucțiuni lansate anterior în execuție, ele fiind confruntate cu operanzii tuturor instrucțiunilor care așteaptă lansarea în execuție pînă ce nu mai intervin conflicte.

Porțiunea aritmetică a secțiunii scalare este formată dintr-o unitate de încărcare/memorare și 5 unități funcționale aritmetice independente care preiau date și depun rezultate într-un set (fișier) de 256 registre pe 64 biți. Unitatea de încărcare/memorare deplasează datele între registre și memoria principală. Perioada ceasului este $\tau = 20$ ns, iar timpii de funcționare ai unităților sînt :

Unitatea funcțională	(ns)	Unitatea de timp (perioade de ceas)
1	2	3
încărcare/memorare	300	15
adunare/scădere	100	5
înmulțire	100	5

1	2	3
operație logică	60	3
ciclu unic	20	1
împărțire, extragerea radicalului, conversie pentru 64 biți	1080	54
pentru 32 biți	600	30

Evaluările reprezintă timpii totali necesari pentru calcularea unui rezultat fie pe 32 biţi, fie pe 64 biţi; toate unităţile de execuţie, cu excepţia ultimei sînt pipeline şi pot prelua un nou set de argumente la fiecare perioadă de ceas. Totuşi, grupul de registre poate furniza cel mult o pereche nouă de argumente într-o perioadă de ceas, acesta fiind factorul cel mai probabil de limitare a vitezei de calcul în secţiunea scalară, în ansamblul ei. Unitatea pentru împărţire, extragere a radicalului şi conversie nu este pipeline, acceptînd argumente noi la fiecare 54 perioade de ceas. Rezultatul produs de oricare unitate poate fi trecut direct la intrarea oricăreia, într-un mod denumit „*shortstopping*". Cînd se poate aplica, acest proces elimină timpul necesar scrierii rezultatelor în registre şi regăsirea lor pentru utilizarea în următoarea operaţie aritmetică. Timpii prezentaţi presupun că procesul shortstopping are loc şi, deci, nu include timpul necesar scrierii rezultatelor în registre sau memorie. Grupul de registre poate furniza cel mult doi operanzi instrucţiunii curente şi memora un rezultat al instrucţiunii anterioare, concurent, în cursul fiecărei perioade de ceas. Este suficient pentru atingerea unei performanţe scalare de 45 Mflop/s, faţă de maximum de o instrucţiune executată la fiecare 20 ns, sau 50 Mflop/s.

Accesul la memoria principală este controlat de către unitatea de încărcare/memorare care lucrează ca un pipeline care poate executa o citire (încărcare-load) din memorie la fiecare perioadă de ceas sau o scriere (memorare-store) în memorie la fiecare două perioade de ceas. Unitatea posedă un bufer pentru cel mult 6 cereri de citire şi 3 pentru scriere. Un cuvînt poate fi citit din memorie şi încărcat într-un registru în 300 ns, dacă memoria nu este ocupată. Dacă unitatea de memorie adresată este ocupată se consumă suplimentar 80 ns.

Operaţiile cu vectori numerici sau şiruri de caractere se execută în secţiunea vectorială care posedă una, două sau 4 unităţi *pipeline în virgulă mobilă* şi o unitate pentru operaţii cu şiruri de caractere (*string unit*), alimentate cu fluxuri de date de o unitate distinctă. Spre deosebire de CRAY X-MP, nu există registre vectoriale, toate operaţiile vectoriale fiind de tipul memorie-principală, datele parcurcînd aproximativ 15 m de fir în comparaţie cu mai puţin de 1,8 la CRAY X-MP. Această diferenţă, explică, în parte, timpul de iniţializare mai mare al calculatorului CYBER 205. Un vector poate avea pînă la 65535 elemente adresate succesiv. Dacă datele necesare nu sînt memorate consecutiv, elementele dorite pot fi selectate cu un *vector de control*, care posedă un bit pentru fiecare cuvînt al vectorului. Apoi, operaţia se execută numai cu elementele pentru care bitul corespunzător de control este 1. Oricum, toate elementele vectorului, memorate consecutiv, trebuie citite din memorie, chiar dacă numai un procent mic din ele vor fi prelucrate. Dacă vectorul de control are puţini biţi de „1", elementele specificate ale unui vector lung pot fi selectate

cu o operație de comprimare (compress) și re-memorate consecutiv. Operațiile care urmează pot fi executate mai eficient, cu noul vector comprimat. În plus, în unitatea de flux sînt implementate cu microcod instrucțiuni de dispersare/grupare. Acestea adresează memoria fie aleatoriu, conform unei liste de indexare, fie periodic (la intervale egale).

Datele sosesc de la memoria principală în trei fluxuri : A și B pentru cele două fluxuri de numere în virgulă mobilă și (M, X, Y) pentru vectorii de control și șiruri de caractere. Există două fluxuri de ieșire : C pentru numere în virgulă mobilă și R pentru șiruri de caractere. Fiecare are lărgimea de 128 biți și este distribuit de unitatea de flux în fluxuri de date de 128 biți pentru prelucrarea în unități pipeline în virgulă mobilă și fluxuri de 16 biți pentru utilizarea în unitatea de prelucrare a șirurilor de caractere. Fiecare unitate pipeline din cele identice (P1 la P4 în fig. 2.16) constă din 5 unități pipeline funcționale separate pentru adunare, înmulțire, deplasare și întîrziere, conectate prin unitatea de interschimbare a datelor (vezi fig. 2.17). Împărțirea și extragerea rădăcinii pătrate sînt executate în unitatea de înmulțire. Fiecare unitate este conectată la unitatea de interschimbare a datelor prin 3 căi de date de 128 biți (2 căi de intrare A și B și o cale de ieșire, C). Pe aceste căi se pot transfera date cu viteze de 100 milioane rezultate pe 64 biți pe secundă, pe unitate (Mr/s). Unitățile propriu-zise pot genera rezultate la jumătate din această viteză (50 Mr/s pentru operații pe 64 biți și 100 Mr/s pentru operații pe 32 biți). Pentru instrucțiuni vectoriale simple care folosesc o singură unitate, unitatea de interschimbare conectează fluxurile de intrare A și B, și fluxul de ieșire C, la unitatea funcțională indicată, realizîndu-se o viteză de calcul asimptotică de 50 Mflop/s (operații pe 64 biți) și 100 Mflop/s (operații pe 32 biți) pentru fiecare unitate pipeline. Dacă două instrucțiuni vectoriale succesive folosesc unități diferite, au un operand scalar și sînt precedate

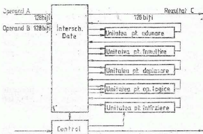


Fig. 2.17 Organizarea de ansamblu a unei unități pipeline de la CDC CYBER 205. Mașina poate avea una, două sau patru astfel de unități pipeline de uz general. (Fotografie furnizată de Control Data Corporation.)

de o instrucţiune de selectare a legăturii, atunci se realizează „unirea" (linkage). Fluxul de ieşire de la prima unitate este transmis de unitatea de interschimbare la intrarea celei de-a doua unităţi. În acest mod, cele două unităţi operează concurent, iar cele doua instrucţiuni vectoriale lucrează ca una singură, fără referinţe intermediare la memoria principală. Iată două exemple de astfel de instrucţiuni triadice (unite) — o instrucţiune triadică are 3 argumente de intrare, de exemplu $A+B\times C$; o operaţie diadică are două argumente, de ex. $A+B$:

vector+scalar*vector, (vector+scalar)*vector,

care sînt folosite frecvent în probleme cu matrici (de exemplu produsul interior a doi vectori prin metoda produsului intermediar. vezi § 5.3.2). Această posibilitate joacă acelaşi rol cu înlănţuirea la CRAY X-MP, dar este mai restrictivă. La CRAY 205 pot fi unite cel mult două operaţii, iar un operand trebuie să fie scalar. Pentru astfel de triade unite performanţa asimptotică a unităţii pipeline în virgulă mobilă se dublează la 100 şi 200 Mflop/s pentru operaţii aritmetice pe 64, respectiv 32 biţi. Deci, performanţa asimptotică maximă este la CYBER 205 la 800 Mflop/s pentru triade unite în cazul operaţiilor aritmetice pe 32 biţi şi al unei maşini cu 4 unităţi pipeline.

Fig 2.18 prezintă mai detaliat organizarea de ansamblu a unităţilor pipeline pentru adunare şi înmulţire (operaţia de adunare (fig. 2.18(a) este împărţită în 7 suboperaţii principale). O conexiune inversă (short-stop) la segmentul ADD permite adunarea unui rezultat ne-normalizat al unui element al unei operaţii vectoriale la următorul element al vectorului. Această posibilitate este folosită de instrucţiunea care produce vectorul $C_{i+1}=C_i+B$; $C_0=A$. Un alt shortstop preia rezultatul normalizat C al unităţii pipeline de adunare şi îl transformă în operandul de intrare B. Rezultatul ajunge înapoi la B opt perioade de ceas după ce operanzii iniţiali au intrat în unitatea pipeline, de unde posibilitatea acumulării $C_{i+8}=C_i+$ $+A_{i+8}$. Această facilitate este folosită la adunarea tuturor elementelor unui vector, ca şi în instrucţiunile de realizare a produsului interior. Unitatea pipeline pentru înmulţire (fig 2.18(b)) are un shortstop similar (întîrziere de 7 perioade de ceas) folosit pentru calcularea produsului tuturor elementelor unui vector. Rata asimptotică este pentru înmulţire de 50 Mflop/s (operaţii artmetice pe 64 biţi) sau 100 Mflop/s (operaţii aritmetice pe 32 biţi) pentru un pipeline, iar pentru împărţire de 4 Mflop/s (operaţii pe 64 biţi) sau 15,35 Mflop/s (operaţii pe 32 biţi) pentru un pipeline. Viteza de execuţie a împărţirii poate fi dublată prin adăugarea unor facilităţi opţionale.

Unitatea de prelucrare a şirurilor execută toate operaţiile logice şi de prelucrare a şirurilor de caractere pe biţi sau pe baiţi. De asemenea prelucrează vectorul de control asociat cu marcarea operaţiilor în virgulă mobilă. Toate căile de date la această unitate sînt pe 16 biţi. Există 2 căi de intrare, X şi Y, şi o cale pentru masca biţilor de control, M. Fluxul de ieşire se notează cu R. La orice CYBER 205 rezultatele operaţiilor logice pe bit se obţin cu viteză de 800 Mbit/s.

Calculatorul CYBER 205 de bază posedă 8 porturi de I/E, pe 32 biţi, cu viteza de transfer de 200 Mb/s. Un al doilea grup de 8 porturi de I/E poate fi adăugat opţional, realizîndu-se în ansamblu, o viteză de transfer

de 3200 Mbit/s. Fiecare canal de I/E are un registru bufer de 4096 de biţi. Toate canalele de I/E partajează încă un registru bufer de I/E care face legătura cu unitatea de interfaţare la memorie printr-o magistrală pentru citirea datelor, pe 128 biţi (R3) şi o magistrală pentru scrieri pe 128 biţi (W2). Rata totală de transfer de I/E este disponibilă unităţii centrale la

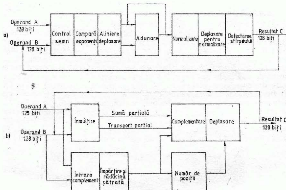


Fig. 2.18 Diagrama bloc a secţiunilor principale pentru (a) sumatorul în virgulă mobilă şi (b) unităţile pipeline pentru înmulţire de la CDC CYBER 205. (Fotografie furnizată de Control Data Corporation).

orice nivel, vectorial sau scalar. Unitatea de control a funcţionării corecte (maintenance control unit — MCU) poate utiliza orice canal de I/E. Această unitate asigură interfaţa utilizatorului pentru verificarea funcţionării, controlului sistemului (inclusiv a fazei de iniţializare) şi conducerii curente. MCU este formată dintr-o unitate de control, o imprimantă, o unitate de disc şi o interfaţă de canal. În regim off-line, MCU încarcă rutine de test de pe disc şi afişează rezultatele execuţiei lor. On-line, MCU supraveghează CPU şi afişează starea acesteia.

Arhitectura calculatorului CYBER 205 poate fi scrisă în notaţia ASN descrisăn în § 1.2.4, ca

$$C(\text{CYBER } 205) = Ipv^{20}_{3,64}[\{Ep,\ P2.P3\}]\overline{_{10,128}}M2\overline{_{4,512}}4MI^{80}_{1M,64}]$$

$$MI(\text{main}) = \{_{512}16M3_{128K,32}\}\ ;\ M3(\text{stack}) = \{_{32}8M^{80}_{16K,32}\}$$

$$Ep(\text{vector}) = \{4\overline{F}p, Bp3\} - Bp4_{5,128}\ ;\ M2(\text{buffer}) = 5M_{1,512}$$

$$Fp = Fp_{32,64}(+.*.\div.\sqrt{})$$

$$Bp3(\text{string}) = B_{16}(\text{bit, byte})\ ;\ Bp4(\text{stream}) = Bp(\text{scatter/gather})$$

$$P2(scalar)=5Epl_{3*64}256M_{1*64\ \ 3*128}$$

$$5Ep1=\{Fp_{64}(+),\ Fp_{64}(*),\ Ep(1\text{-}cycle),\ Bp_{64}(logical),\ Fp_{64}(\sqrt{},\ \div)\}$$

$$P3(I/O)=16\{D_{\overline{32}}IO_{\overline{32}}M_{\overline{32}}\}_{\overline{32}}M_{32*128}(I/O\ buffer)_{\overline{2*128}}$$

2.3.3. Tehnologia

Interesul privind tehnologia calculatorului CYBER 205 se concentrează asupra unui nou circuit LSI, a tehnologiei de încapsulare și răcire pentru schemele hardware care folosesc masive de porți logice ECL bipolar. În fig. 2.19 se arată o placă cu 15 straturi, care poate avea 10×15 circuite LSI. Principala caracteristică este conducta cu freon, care traversează de 10 ori placa. Circuitele LSI sînt montate pe socluri ceramice, fixate direct pe conductele de răcire, care mențin o temperatură de 55 ± 1°C. În fig. 2.20 se prezintă înlocuirea unui circuit LSI. În fig. 2.21 apar diversele conectoare și socluri folosite pentru fixarea circuitelor, iar în fig. 2.22 se prezintă modul lor de asamblare. Tabletele de cupru fac contact termic direct cu conducta de răcire. Circuitul LSI are 52 conexiuni externe plasate lateral și în partea inferioară a capsulei (denumită masiv LSI, fig. 2.22 *stînga*). Doi conectori, de fiecare parte a capsulei, pentru 25 pini fie-

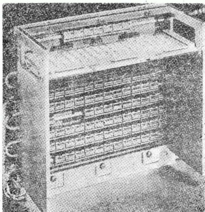


Fig. 2.19 O placă logică de la CDC CYBER 205. Freonul circulă prin conducte pe care sînt plasate circuitele logice. LSI. Fiecare placă cu 15 straturi poate avea cel mult 150 circuite LSI. (Fotografie furnizată de Control Data Corporation).

care, realizează legătura electrică cu placa (fig. 2.21 *centru* şi fig. 2.22 *dreapta*), ei fiind într-un înveliş plastic. Masivul LSI este fixat în soclu cu un clip. Pentru a ne forma o idee a comprimării obţinute cu circuitele LSI, menţionăm că întreaga secţiune scalară L (vezi fig. 2.14 şi 2.15) ocupă 16 plăci, aflate într-un dulap de aproximativ 2.1 m lungime. Circuitele LSI

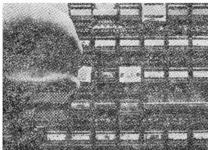


Fig. 2.20 O imagine de detaliu a unei plăci cu circuite de la CDC CYBER 205, arătînd cum un tehnician înlocuieşte un circuit LSI, aflat pe soclu ceramic. Barele orizontale sînt conductele de răcire. (Fotografie furnizată de Control Data Corporation).

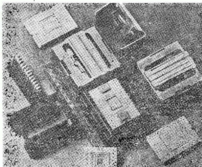


Fig. 2.21 Diversele socluri, conectorii şi fixatori folosite pentru cuplarea circuitelor LSI la conductele de răcire. În centru, se observă două circuite montate într-o secţiune a plăcii logice (Fotografie furnizată de Control Data Corporation).

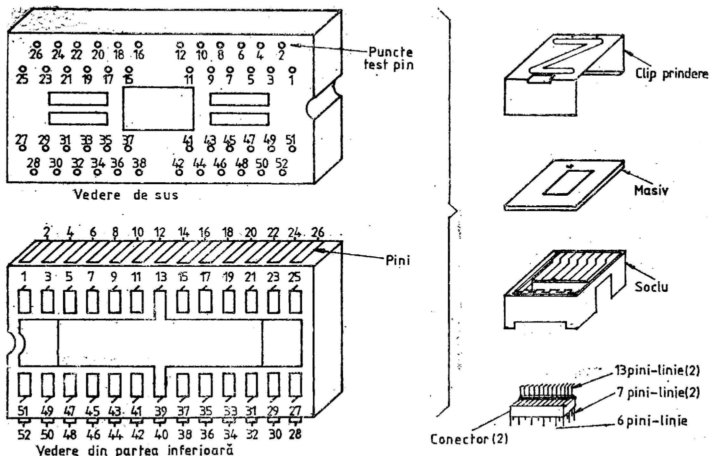


Fig. 2.22 Stînga Partea superioară și cea inferioară a unui circuit LSI montat pe soclul lui ceramic. Circuitul este denumit masiv LSI. Dreapta : cum se atașează masivul LSI la conductele de răcire și placa logică. (Imagini furnizate de Control Data Corporation.)

folosesc tranzitori bipolari ECL. În fiecare capsulă se află un masiv de aproximativ 168 comutatoare ECL (echivalentul a 300 porţi). Timpii de propagare sînt mai mici de 1 ns, iar produsul putere × timp de propagare este de aproximativ 6pj. Se folosesc numai 29 tipuri de circuite LSI. Prin utilizarea tehnologiei LSI, se reduce consumul de putere şi se îmbunătăţesc condiţiile de întreţinere. Prin reducerea numărului de conexiuni externe ale circuitelor, fiabilitatea este îmbunătăţită de aproximativ 6 ori. Puterea consumată de fiecare circuit LSI este mai mică de 10 ori decît cea cerută de tehnologia SSI anterioară.

Memoria principală a seriei 400 foloseşte circuite de memorie bipolară de 4 K cu un timp de acces de 80 ns. Se mai folosesc circuite ECL 100 K. Fiecare milion de cuvinte de 64 biţi ocupă 2 secţiuni de memorie, fiecare format din 8 stive, aşa cum se arată în fig. 2.23. Fig. 2.24 reprezintă o imagine de detaliu a unei stive care memorează 128 K jumătăţi de cuvinte de 32 biţi în 8 blocuri independente de memorie. O stivă este formată din 2 plăci de intrare, o placă de ieşire şi 16 plăci de memorie. Între plăci se află conductele sistemului de răcire. O placă de memorie, fig. 2.25, asigură în paralel 20 sau 19 biţi ai unui cuvînt, pe baza circuitelor de 4 K. O pereche de plăci de memorie formează un bloc care accesează în paralel 39 biţi (32 biţi de date şi 7 biţi SECDED), cîte un bit de la fiecare din cele 39 capsule.

Deşi organizarea memoriei este identică la seria 600 cu cea de la seria 400, din punctul de vedere al utilizatorului, tehnologia şi modul de încapsulare sînt diferite. Se folosesc circuite MOS de 16 Kb, nivelul mai mare de integrare făcînd posibile configuraţiile cu 1, 2, 4, 8, 12 sau 16 Mw. La maşina cu 16 Mw, fiecare din cele 8 secţiuni de memorie con-

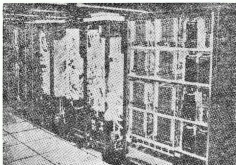


Fig. 2.23 Dulapurile unde se află memoria de un milion de cuvinte pe 64 biţi de la CYBER 205. La extreme se află două secţiuni ale memoriei, iar în centru se află, în două dulapuri mai mici, interfaţa. În secţiunea de memorie din dreapta se pot vedea cele 8 stive de memorie. CYBER 205 poate avea 1, 2 sau 4 Mw.

ține 2 Mw, organizate în 8 module a 512 K jumătăți de cuvinte de 32 biți. Fiecare modul joacă același rol cu stiva de la seria 400 și este format dintr-o placă de control cu circuite ECL de 100 K și un ansamblu de plăci de memorie pe care se află montate 4×4 plăci mai mici. Acestea joacă rolul celor 16 plăci ale stivei de la seria 400, fiecare conținînd 20 circuite MOS de la 16 Kb, fiind identice cu cele de la seria 400, din punctul de vedere al utilizatorului, tehnologia și modul de încapsulare fiind echivalentul maximului de 4 Mw de la seria 400. La primul strat se mai pot adăuga suplimentar încă 3. Fiecare contribuie cu încă 4 Mw de locații consecutive de memorie, formînd memorii de 8.12 și 16 Mw. Timpul de acces la seria 600 este tot de 80 ns.

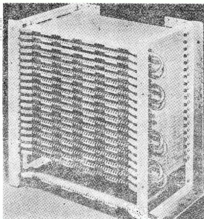


Fig. 2.24 O stivă de memorie de la CYBER 205. Stiva are două plăci pentru intrare, una pentru ieșirea și 16 plăci cu circuite de memorie. O pereche de plăci de memorie conține un bloc de 16K jumătăți de cuvinte pe 32 biți. Cele 8 blocuri ale unei stive dau un total de 128K jumătăți de cuvinte. (Fotografie furnizată de Control Data Corporation).

Funcționarea secțiunilor scalară și vectorială este controlată prin microcod aflat într-o memorie auxiliară, construită cu pînă la 90 circuite ECL de 100K. O astfel de placă apare în fig. 2.19. Aceste memorii, stivă pentru 128 instrucțiuni pe 32 biți și cele 256 registre de 64 biți se află pe plăci auxiliare, fiind formate din circuite ECL de 100K. Ciclul de citire/scriere al acestor elemente de memorie de 10 ns include timpul de propagare pe poartă al circuitelor ECL de 100K, de 1 ns.

2.3.4. Setul de instrucțiuni

Setul de instrucțiuni al calculatorului CYBER 205 este în mod special bogat în posibilități, dar este necesar să analizăm sistemul de adresare și formatele aritmetice. CYBER 205 are un sistem de adresare virtuală. Cîmpul de adresă virtuală al unei instrucțiuni are 48 biți și reprezintă adresa unui bit individual în memoria virtuală. Astfel, se pot adresa pînă

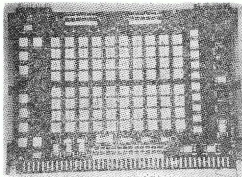


Fig. 2.25 O placă de memorie de la CYBER 205, care furnizează 20 biți în paralel. Circuitele de 4K biți sînt montate central în două matrici 4×10. Patru circuite, sînt asociate fiecărei poziții binare din cuvînt, rezultînd un total de 16K adrese de memorie. Două astfel de plăci formează un bloc de memorie și asigură cei 39 biți ai unei jumătăți de cuvînt (32 pentru date și 7 biți SECDED). (Fotografie furnizată de Control Data Corporation.)

lq $2,8\times10^{14}$ biți, $3,5\times10^{13}$ baiți, $8,8\times10^{12}$ jumătăți de cuvinte pe 32 biți sau $4,4\times10^{22}$ cuvinte pe 64 biți de memorie virtuală. Jumătatea superioară a memoriei virtuale este rezervată pentru sistemul de operare și vectori temporari, lăsînd spațiul de adresare virtuală de $2,2\times10^{12}$ cuvinte pe 64 biți pentru programele utilizator. Pe de altă parte, memoria principală fizică are maximum $4,2\times10^{6}$ cuvinte pe 64 biți. Sistemul de operare transferă programe și date în memoria principală, fie ca pagini mici (de 512, 2K sau 8K cuvinte de 64 biți) fie ca pagini mari de 64 K cuvinte de 64 biți. Secțiunea scalară folosește 16 registre asociative pentru translatarea adreselor virtuale în adrese fizice. Registrele păstrează cuvinte asociative care conțin adresele virtuale, și cele fizice corespunzătoare celor mai recent folosite 16 pagini. Toate pot fi comparate în cursul unei perioade de ceas. Dacă adresa virtuală nu se găsește, comparația va continua într-o tabelă care conține o extensie a listei cuvintelor asociative în memoria principală. Dacă este găsită, adresa virtuală este translatată

în adresa fizică, iar execuția programului continuă. Dacă pagina nu este găsită, starea programului este reținută automat în memorie și se intră în programul monitor pentru a se transfera controlul altui job.

Operațiile aritmetice în virgulă mobilă pot fi executate fie cu jumătăți de cuvinte de 32 biți, fie cu cuvinte de 64 biți. Numerele sînt exprimate ca $C \times 2^E$, unde coeficientul C și exponentul E sînt întregi în complement față de doi. În formatul pe 32 biți, E are 8 biți, iar C are 24, astfel că numerele aparțin domeniului $\pm 10^{-27}$ la $\pm 10^{+40}$. În formatul pe 64 biți, E are 16 biți, iar C are 48, numerele luînd valori în domeniul $\pm 10^{-8616}$ la $\pm 10^{+8644}$. Punctele binare se află în extrema dreaptă a cîmpurilor binare ce reprezintă atît pe C, cît și pe E, iar bitul de semn la extrema stîngă. Un număr este normalizat cînd bitul de semn al coeficientului este diferit de bitul vecin din dreapta. Rezultatele în dublă precizie se memorează ca două numere cu același format ca un rezultat în simplă precizie, și adresate ca partea superioară, respectiv inferioară, a rezultatului. Ele pot fi prelucrate separat. O altă caracteristică este asigurarea numărului de cifre semnificative egal cu cel al operandului mai puțin semnificativ. În acest mod, rezultatul unei operații în virgulă mobilă este deplasat astfel încît numărul cifrelor semnificative să egaleze pe cele ale operandului mai puțin-precis.

La CYBER 205 instrucțiunile au trei adrese și pot fi pe 32 biți sau 64 biți, cu 12 formate posibile. Există 219 instrucțiuni diferite care pot fi împărțite în următoarele categorii (în paranteză se dă numărul de instrucțiuni din fiecare categorie):

Registru (60)	Macro vectorial (15)
Index (9)	Sir (1)
Salt (29)	Sir logic (8)
Vector (28)	Netipic (51)
Vector rar (11)	Monitor (7)

În această scurtă trecere în revistă nu vom încerca să descriem toate instrucțiunile ci numai posibilitățile setului de instrucțiuni prin exemplificări ale celor mai interesante dintre ele.

Instrucțiunile cu registre manipulează date în grupul de 256 registre pe 64 biți, fie ca jumătăți de cuvinte pe 32 biți, fie ca cuvinte pe 64 biți, în funcție de instrucțiuni. R, S și T notează numere de registre reprezentate pe 8 biți. Într-un cuvînt, biții sînt numerotați de la stînga la dreapta începînd cu zero. Iată cîteva exemple:

ADDX R, S, T	adună partea de adresă (biții 16 la 63) a registrului R la registrul S și memorează în registrul T
EXPH R, T	ia jumătatea de cuvînt exponent din registrul R și plasează-l în pozițiile binare mai puțin semnificative ale registrului T.

Instrucțiunile index încarcă și manipulează porțiuni de 16, 24 sau 48 biți ai registrelor.

IS R, 116	crește cei mai semnificativi 48 biți ai registrului Ru operandul I16 pe 16 biți din poziția 16 la la 31 ai codului instrucțiunii.

Instrucţiunile de salt pot fi folosite pentru compararea sau examinarea unor biţi, a unor indici pe 48 biţi, a operanzilor pe 32 sau 64 biţi. Rezultatul comparaţiei determină dacă se continuă secvenţa sau se efectuează salt la alte secvenţe de instrucţiuni:

CFPEQ A, X, [B, Y] Se verifică egalitatea numerelor în virgulă mobilă pe 64 biţi din registrele A şi X. Dacă comparaţia are succes, se efectuează salt la locaţia specificată de conţinutul lui [B, Y].

Instrucţiunile vectoriale execută operaţii cu seturi ordonate de numere memorate în locaţii succesive de memorie. Operaţia se execută element cu element şi rezultatul se memorează într-un set consecutiv de locaţii. Vectorul poate avea cel mult 65535 elemente. Un vector este specificat într-o instrucţiune prin numerele (pe 8 biţi fiecare) unei perechi de registre, de exemplu [A, X] sau [01, 02]. Primul registru conţine adresa de bază (48 biţi) şi lungimea cîmpului (16 biţi) ale vectorului, iar al doilea conţine deplasarea pe 16 biţi faţă de adresa de bază, unde începe vectorul. De asemenea, o instrucţiune vectorială include numărul registrului (8 biţi) care conţine adresa de start a vectorului de control (48 biţi). Vectorul de control este un vector binar ce conţine cîte un bit pentru fiecare operand vectorial. Este folosit pentru controlul (sau mascarea) memorării rezultatului operaţiei vectoriale. Se poate cere, de exemplu, ca memorarea să aibă loc numai pentru elementele corespunzătoare biţilor de control 1 (sau alternativ, zero). În continuare A, B, C, X, Y, Z notează registrele în domeniul 00 la FF în hexazecimal.

MPYUV [A, X], [B, Y], C, Z Înmulţeşte vectorul specificat de registrul [A, X] cu vectorul specificat de [B, Y] conform vectorului de control specificat de registrul Z. Vectorul rezultat este specificat de registrul C. Deplasarea pentru rezultat se află, prin convenţie, în registrul C+1.

Instrucţiunile vectoriale includ adunarea, scăderea, înmulţidrea, împărţirea (folosind fie partea superioară, fie inferioară a rezultatelor de lungime dublă, normalizate sau conform procedurii de păstrare a cifrelor semnificative), funcţii APL (Iverson 1962), modificarea reprezentării numerelor între formatele în virgulă mobilă pe 64 sau 32 biţi, radical, trunchierea, împachetarea şi despachetarea coeficientului şi exponentului numerelor în virgulă mobilă.

Dacă un vector conţine multe elemente nule sau nenule, se poate folosi un format special. Un astfel de vector poate fi comprimat într-un vector rar definit de un vector de ordine şi un vector pentru date. Primul este un vector binar cu 1 bit pentru fiecare bit al vectorului complet. Prezenţa unui element nenul este identificată de un bit unu, iar prezenţa unui element nul, de un bit zero. Cu toate poziţiile elementelor nenule identificate de vectorul de ordine, este necesar ca vectorul pentru date să memoreze numai valorile elementelor nenule în ordinea în care apar în vectorul complet. Un vector rar este specificat într-o instrucţiune prin numerele unei perechi de registre. Primul conţine adresa de bază şi lungi-

mea cîmpului vectorului de ordine. Cu vectori rari se pot executa operaţii de adunare, scădere, înmulţire şi împărţire, de exemplu:

ADDINS [A, X], [B, Y], [C, Z] adună normalizat vectorul rar specificat de registrele [A, X] la vectorul rar specificat de [B, Y], memorînd rezultatul ca vector rar specificat de registrele [C, Z].

Vectori rari pot fi produşi de instrucţiuni de comparare între vectori:

CMPEQ [A ,X], [B, Y], Z compară şi formează vectorii de ordine: dacă $A_n = B_n$, bitul Z_n va fi egal cu 1, altfel $Z_n = 0$, urmată de o operaţie de comprimare vectorială, de exemplu:

CPSV A, C, Z comprimă vectorul C, conform vectorului de ordine Z.

Un vector rar poate fi transformat într-un vector complet, de ex:

MRGV A, B, C, Z unifică vectorul A cu vectorul B sub controlul vectorului Z. Dacă B este un vector cu toate elementele nule, atunci vectorul rezultat C este forma extinsă a vectorului rar A:

dacă $Z_n = 1$, C_n = următorul element al lui A;

dacă $Z_n = 0$, C_n = următorul element al lui B.

Instrucţiunile vectoriale de comparare, comprimare, unificare şi mascare sînt exemple de instrucţiuni netipice. În această categorie mai intră citirea ceasului de timp real, numărarea biţilor dintr-un cîmp, simularea defectării, aflarea elementului maxim sau minim al unui vector, unificarea şirurilor de biţi sau baiţi, căutarea unui anumit bait.

Un alt set de instrucţiuni de un interes special sînt macroinstrucţiunile vectoriale, care execută cu o instrucţiune unele din cele mai frecvente operaţii în analiza numerică. Ele sînt implementate în microcod şi execută operaţii care ar solicita în mod normal o subrutină. În toate cazurile, elementele implicate în operaţie pot fi selectate cu un vector de control Z. Iată exemple:

ADJMEAN [A, X], C, Z adiacent înseamnă: $C_n = (A_{n+1} + A_n)/2$

AVG [A, X], [B, Y], C, Z media $C_n = (A_n + B_n)/2$

DELTA [A, X], C, Z diferenţierea delta sau numerică: $C_n = (A_{n-1} - A_n)$

DOTV [A, X], [B, Y], C, Z produsul de lungime dublă ($\Sigma A_n B_n$) memorat în registrele C şi C+1.

VREVV [A, X], C, Z transmite vectorul A în C cu elementele în ordinea inversă.

Operaţiile de dispersare şi grupare aleatoare sau periodică (vezi § 2.2.4) se execută cu instrucţiuni unice la CYBER 205. Ele se aplică elementelor sau grupurilor deplasate la sau de la memoria principală sau grupul de registre. Exemple:

VTOVX [A, X], B, C transmisie vector la vector indexat, B→C indexat de A.

VXTOV [A, X], B, C transmisie de vector indexat la vector, B indexat de A→C

Listele indexate folosite mai sus pot fi generate în orice modalitate convenabilă; pentru acest scop se poate folosi o instrucţiune de căutare, de exemplu:

SRCHEQ A, B, C, Z — Căutarea egalităţii şi formarea listei indexate. Se compară A_n cu toate elementele lui B pînă se obţine egalitate. Numărul comparaţiilor fără succes înregistrate înainte de egalitate este memorat de C_n. Se repetă pentru toate elementele lui A. Numerele C_n sînt de fapt indicii elementelor ce satisfac condiţia de egalitate. Comparaţia poate fi limitată la anumite elemente prin vectorul de control Z.

O altă formă de instrucţiune de căutare asigură un index unic, de exemplu:

SELLT [A, X], [B, Y], C,Z — selectează conform condiţiilor mai mic ca: elementele corespunzătoare ale lui A şi B. se compară începînd cu primul element. Indexul primei perechi care satisface condiţia (aici $A_n < B_n$) este plasat în registrul C. Perechile de elemente sînt rărite sau incluse funcţie de biţii vectorului de control Z.

Instrucţiunile cu şiruri execută operaţii cu şiruri de date sub forma baiţilor. Acestea pot fi caracterele dintr-o mulţime posibilă de 256, inclusiv standardele ASCII şi EBCDIC. Spre deosebire de CYBER 203, CYBER 205 implementează numai o astfel de instrucţiune orientată pe bait:

MOVL [A, X], [C, Y], I8 — Baiţii din A sînt deplasaţi la stînga, devenind baiţii lui C. Copii ale baitului I8 sînt introduse la dreapta la cerere

Pe de altă parte, operaţiile logice cu şiruri se execută pe bit. Se pot executa cu instrucţiuni diferite 8 operaţii logice, de exemplu:

AND [A, X], [B, Y], [C, Z] — operaţia logică SI a biţilor lui A cu biţii lui B, rezultatul fiind în C.

NOR [A, X], [B, Y], [C,Z] — negarea operaţiei SAU a biţilor lui A cu biţii lui B, formîndu-se rezultatul în C.

Instrucţiunile monitor pot fi folosite numai în modul monitor. Ele sînt folosite de sistemul de operare pentru încărcarea şi memorarea cuvintelor asociative la adrese absolute în memorie, asigurînd buna funcţionare a sistemului de memorie virtuală şi gestiune a întrepruperilor. Aceste instrucţiuni nu pot fi executate de un program utilizator fără a provoca un defect.

2.3.5. Software

Soft-ul dezvoltat pe CYBER 205 este o dezvoltare a celui proiectat pentru CDC STAR 100 şi funcţional pe STAR 100, CYBER 203, şi CYBER 205 începînd cu anul 1974. Principalele elemente sînt:

(1) CYBER 205—OS—un sistem de operare batch şi interactiv;

(2) CYBER 200 FORTRAN — un compilator vectorizant pentru principalul limbaj de nivel înalt;

(3) CYBER 200 META — limbajul de asamblare care asigură accesul la toate resursele hardware ale mașinii;

(4) utilitarele CYBER — inclusiv un încărcător (loader), editor de fișiere și programe de întreținere.

Sistemul de operare CYBER 200 este proiectat să asigure acces batch și interactiv, fie local, fie de la distanță, via un calculator front-end din seria CYBER 180, IBM sau VAX. Memoria de masă este asigurată de unitățile de disc CDC 819 (capacitate 4800 Mb, viteza medie de transfer 36,8 Mb/s, timp de poziționare mediu de 50 ms) conectate la canalele calculatorului CYBER 205. Fiecare utilizator poate folosi o memorie pînă la $2{,}2 \times 10^{12}$ cuvinte de 64 biți. Programe care folosesc acest spațiu sînt memorate pe disc și transferate în memoria principală a calculatorului (maximum 4×10^{6} cuvinte de 64 biți) în pagini. Sistemul de operare are sarcina de a aloca memoria fizică în pagini de dimensiune corespunzătoare și de a le distribui programelor utilizator, care pot fi executate într-un mediu multiprogram. Pentru aceasta sistemul de operare folosește instrucțiuni ale modului monitor pentru schimbarea cuvintelor asociative între registrele asociate și tabela paginilor. Sistemul de operare este modular, iar comunicația între diferitele părți se realizează prin mesaje. Nucleul sistemului de operare este memorat în memoria virtuală și paginat în memoria principală cînd este necesar. Numai modulul *Kernel* și *paginatorul* sînt rezidente în memorie permanent. Modulul Kernel alocă timpul diferitelor job-uri active și transmite mesaje părților diferite ale soft-ului. Paginatorul gestionează alocarea memoriei și înlocuirea paginilor.

O altă parte a sistemului de operare, denumită sistemul virtual asigură introducerea job-urilor de la terminalele batch sau interactive și îndepărtează job-urile terminate sau inactive. Mesajele job-urilor active pentru operații de I/E sînt prelucrate de sistemul virtual și trimise rețelei de cuplare slabă (loosely coupled network — LCM) pentru execuție. Se execută și task-uri de evidență. Un program *operator* asigură comunicarea interactivă cu operatorul care poate: afișa job-urile utilizator și informația asociată; încheia sau suspenda job-urile; analiza tabelele sistem; și controla cursul job-urilor prin calculator.

Compilatoarele CYBER 200 FORTRAN includ atît standardele ANSI X3.9—1966 și 1978, cu extensii ce permit utilizatorului să folosească posibilitățile hardware ale calculatorului CYBER 205. S-au introdus noi posibilități pentru conformare la standardul ANSI X3.9—1978 și care a devenit un standard Control Data la compilatoarele CDC anterioare (NAMELIST I/O, ENCODE/DECODE, BUFFER IN și BUFFER OUT). Compilatorul are un vectorizator automat care înlocuiește buclele DO cu instrucțiuni vectoriale sau rutine STACKLIB cînd astfel de substituții nu modifică logica problemei. În plus, compilatorul CYBER 200 FORTRAN posedă un *optimizator* care replanifică ordinea instrucțiunilor scalare pentru a optimiza utilizarea registrelor scalare și a unităților funcționale pipeline scalare, asigurînd astfel maximum concurenței funcționale fără intervenția specifică a utilizatorului.

Operațiile vectoriale sînt specificate prin descriptori care definesc vectorii. Ele se translatează direct în formatul instrucțiunii mașinii, de scris în §2.3.4. Un vector este definit de un nume de masiv, un index de start

şi de lungime. Astfel, fiind dat un masiv A, A(10 ; 100) înseamnă vectorul care începe la locaţia A(10) şi are 100 elemente. Se pot folosi descriptori implicaţi în expresii, sau declaraţii dinamice. Iată exemple :

```
      DIMENSION A(1000), B(1000)
      DESCRIPTOR AB, BD
C(1) BY DECLARATION
      ASSIGN AD, A(1 ; 599)
      ASSIGN BD, B (2 ; 999)
      AD=BD*2.0
C(2) IMPLICIT USE OF DESCRIPTORS
      A(1 ; 999)=B(2 ; 999)*2.0
```

În ambele cazuri masivul de elemente B(2),. . .,B(1000) este înmulţit cu 2 şi memorat în locaţiile A(1), . . .,A(999). În cazul (1) în instrucţiunea aritmetică numele masivelor sînt înlocuite de descriptori, iar în cazul (2) elementele masivului ce urmează să fie folosite sînt specificate în instrucţiunea aritmetică.

Se poate obţine accesul la toate instrucţiunile calculatorului CYBER 205 prin apeluri de subrutine speciale, sub forma CALL Q8ADDX (R, S, T) care, de exemplu, generează instrucţiunea maşină ADDX R, S, T. Mnemonicele pentru alte instrucţiuni maşină pot primi în mod similar prefixul rezervat Q8 şi folosite ca apeluri de subrutine pentru a genera o singură instrucţiune în locul codului FORTRAN necesar subrutinei. Codul de asamblare poate fi încorporat într-un program FORTRAN printr-o referinţă externă în programul FORTRAN la codul produs de asamblorul CYBER 200, în cursul încărcării părţilor programului.

Unele operaţii vectoriale diadice sau triadice frecvente, inclusiv recursive, au fost programate eficient şi sînt accesibile prin apeluri la subrutine speciale din **STACKLIB**. Iată exemple din cele 25 forme generale.

(*1*) *Adunare recursivă V1*

```
      CALL Q8A010 (A(2), B(2), A(1), N-1)
```

echivalent cu

```
      DO 1 I=2, N
    1 A(I)=B(I)+A(I-1)
```

(*2*) *Adunare cu înmulţire*

```
      CALL Q8MA400 (A(2), C, B(2), D (2), N-1)
```

echivalent cu

```
      DO 1 I=2,.N
    1 A(I)=C*B(I)+D(I)
```

(*3*) *Scădere cu înmulţire, recursiv V1, ordine inversă*

```
      CALL Q8SM013 (A(N-2), B(N-2), C(N-2), A(N-1), N-1)
```

echivalent cu

```
      DO 1 I=2, N
      J=(N+1)-I
    1 A(J)=B(J)-(C(J)*A(J+1))
```

Literele ce urmează lui Q8 identifică tipul operaţiilor, iar codul numeric indică dacă operanzii sînt scalari sau vectoriali şi care operanzi sînt recursivi.

Programul asamblor **META** generează cod binar relocatabil plecînd de la mnemonice de instrucțiuni mașină, proceduri, funcții și directive. Se asigură accesul la toate resursele hardware ale mașinii. Directivele permit programatorului să controleze procesul asamblării. Asamblorul permite: asamblarea condițională, generarea codului re-entrant, care poate fi folosit simultan de mai mulți utilizatori fără duplicarea codului, posibilitatea refolosirii unora sau a tuturor mnemonicelor, posibilitatea de a defini un simbol pentru o mulțime (sau listă) de date, atribuite unor astfel de mulțimi (tipul și numărul elementelor etc.) pot fi stabilite și referite de programator. Procesul de asamblare are o loc în 2 etape (passes). La prima trecere, se interpretează toate instrucțiunile, se atribuie valori simbolurilor și se rezervă locații pentru fiecare instrucțiune. La a doua trecere, se satisfac referințele externe și interne, se generează datele și se produc listing-urile binare de ieșire și de asamblare. În formă programele asamblate sînt modulare și pot fi formate din mai multe subprograme unite de programul **LOADER**.

Programul **LOADER** este un utilitar al sistemului de operare. Preia codul binar relocatabil produs de compilatorul FORTRAN al asamblorului **META**, îl leagă cu rutinele eventual cerute și produce un fișier direct executabil. Utilizatorul controlează caracteristicile fișierului avînd posibilitatea de exemplu, să ceară încărcarea anumitor rutine ca un grup fie într-o pagină mică, fie mare. Fișierele sursă ale sistemului soft, inclusiv compilatorul și asamblorul, ca și programele utilizator sînt memorate ca imagini cartelă în fișiere program ce pot fi create, editate și menținute în modul cartelă-cu-cartelă, de către programul **UPDATE**. Fișierele binare obiect pot fi editate cu editorul bibliotecii obiect.

Activitățile de întreținere a fișierelor, de pregătire a job-ului și de intrare/ieșire sînt executate de calculatorul front-end, lăsînd astfel calculatorul CYBER 205 să îndeplinească sarcina sa principală, de execuție a calculelor de volum mare. O legătură hardware între calculatorul front-end și CYBER 205 este gestionată software, permițîndu-se astfel ca mai multe calculatoare front-end să lucreze concurent.

2.3.6. Performanța

Vom considera în primul rînd performanță lui CYBER 205 în cazul cel mai favorabil, cînd elementele succesive ale tuturor vectorilor sînt memorate contiguu în memorie. La pagina 184 se tratează performanța cînd vectorii nu sînt memorați contiguu. Tabela 2.4 prezintă performanța așteptată pentru o serie de operații aritmetice în virgulă mobilă pe 64 biți în două unități pipeline cu vectori memorați contiguu. Observăm imediat că $n_{1/2}$ este, cu excepția operațiilor de dispersare și grupare, apropiat de valoarea 100; adică, cel puțin de două ori mai mare decît la CRAY X-MP. Deoarece valoarea lui $n_{1/2}$ determină care este cel mai bun algoritm de folosit (vezi cap. 5), se pot folosi algoritmi diferiți pe cele 2 mașini, chiar dacă ambele sînt în categoria calculatoarelor vectoriale pipeline. Pentru cele mai multe instrucțiuni, r_∞ este 100 Mop/s, în comparație cu 70 Mflop/s pentru dispersare/grupare, max/min și instrucțiuni de produs de elemente.

În ultimul paragraf am comparat performanța calculatorului CYBER 205 cu două unități pipeline cu CRAY X-MP pentru operații aritmetice cu aceeași precizie, 64 biți. Există 2 modalități de îmbunătățire a performanței la CYBER 205. O posibilitate este scăderea preciziei la 32 biți în virgulă mobilă, iar cealaltă creșterea numărului de unități pipeline de

Tabelul 2.4 **Performanța vectorială estimată pentru un CYBER 205 cu două unități pipeline pentru o serie de instrucțiuni selectate (care lucrează pe 64 biți), interpretate în termenii r_∞ și $n_{1/2}$ Pefrormanța reală percepută într-un mediu de lucru multiprogramat poate fi diferită, N = numărul elementelor vectorului produs, I = numărul elementelor vectorului (vectorilor) operanzi. Toți vectorii sînt memorați în locații succesive în memorie.**

Instrucțiune	Timp (perioade de ceas)	r (Mop/s)	$n_{1/2}$
Adunare vectorială	51 + 0,5N	100	102
Înmulțire vectorială	52 + 0,5N	100	104
Adunare vectorială rară	88 + 1/16+7N/16	⩽ 89*	⩽ 156
Înmulțire vectorială rară	88 + I/16+7N/16	⩽ 73**	⩽ 128
Produs scalar	116 + I	100	116
Produs de elemente	126 + I	50	126
Max sau min	85 + I	50	86
Comprimare	52 + 0,51	100	104
Mascare sau unificare	56 + 0,5N	100	112
Distribuire aleatoare	83 + 1,25N	40	66
Grupare aleatoare	69 + 1,2N	40	55
Relațională vectorială	56 + 0,5N	100	112

* $I \leqslant 2N$
** $I \leqslant 4N$

la 2 la 4. Ambele operații au efectul dublării numărului de rezultate obținute într-un anumit interval de timp. Executînd simultan aceste operații, se obțin de 4 ori mai multe rezultate în același interval de timp. Dacă pentru producerea a n rezultate pe 64 biți, cu o mașină cu 2 unități pipeline, este necesar timpul

$$t = r_\infty^{-1}(n + n_{1/2}) \tag{2.9}$$

atunci acesta este timpul necesar prelucrării unui vector de lungime $n' = cn$, unde $c=2$ sau 4 în situațiile de mai sus. Înlocuind în ecuația (2.9), obținem

$$t = r_\infty^{-1}(n'/c + n_{1/2}) \tag{2.10a}$$

$$= (cr_\infty)^{-1}(n' + cn_{1/2}) \tag{2.10b}$$

În noua situație (indicată de semnul ') obținem, prin definiție

$$t = (r'_\infty)^{-1}(n' + n'_{1/2}) \tag{2.10c}$$

și de aici prin comparație cu ecuația (2.10b)

$$r'_\infty = c \cdot r_\infty, \quad n'_{1/2} = c \cdot n_{1/2} \tag{2.10d}$$

Astfel, dublînd sau crescînd de 4 ori lungimea vectorului prelucrat într-un anumit timp, se dublează sau crește de 4 ori atît r_∞ cît și $n_{1/2}$. Acest efect poate fi observat în rezultatele furnizate de Kascic (1975) în fig. 2.26, unde se reprezintă timpul pentru adunarea vectorială diadică sau înmulțirea element cu element, operații de forma **C** = **A** op**B** pe un CYBER 205

cu 2 unități pipeline și reprezentare pe 64 biți (curba A : r_∞ = 100 Mflop/s, $n_{1/2}$ = 100) și aceeași operație pe un CYBER 205 cu 2 unități pipeline, dar reprezentare pe 32 biți sau pe un CYBER 205 cu 4 unități pipeline și reprezentare pe 64 biți (curba B : r_∞ =200 Mflop/s, $n_{1/2}$ =200).

Fig. 2.26 mai prezintă curba C, a timpului pentru o operație triadică legată, cum este

$$D = A + B*c \tag{2.11}$$

unde vectorul B este multiplicat cu scalarul c și apoi adunat element cu element la vectorul *A*. Deoarece prezența cea mai mare a overhead-ului care contribuie la valoarea lui $n_{1/2}$ este asociată cu citirea și scrierea numerelor în memoria principală, dublarea lungimii unității pipeline nu modifică semnificativ $n_{1/2}$. Deoarece după execuția a două operații rezultatul este returnat memoriei, r_∞ se dublează. Cu alte cuvinte : fie timpul de execuție pentru o operație vectorială de înmulțire sau adunare

$$t = \tau(n + s + l - 1) \tag{2.12a}$$

unde s este timpul pentru operații de citire sau scriere în memorie, iar l este lungimea unității aritmetice pipeline. La CYBER 205, $S \gg l > 1$, deci

$$r_\infty = \tau^{-1}, \; n_{1/2} = s + l - 1 \approx s \tag{2.12b}$$

Dacă cele două operații sînt unite împreună, timpul de execuție al unei operații vectoriale devine

$$t = \tau(n + s + 2l - 1)/2 \tag{2.13a}$$

deci,

$$r_\infty = 2 \cdot \tau^{-1}, \; n_{1/2} = s + 2l - 1 \approx s \tag{2.13b}$$

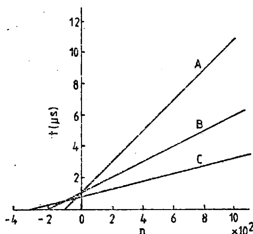


Fig. 2.26 Curbele folosite pentru determinarea lui r_∞ și $n_{1/2}$ la CDC CYBER 205, luate din lucrarea lui Kascic (1979). A, un CYBER 205 cu două unități pipeline, pentru operații aritmetice diadice pe 64 biți ; B, un CYBER 205 cu patru unități pipeline și operații pe 64 biți sau un CYBER 205 cu două unități pipeline pe 32 biți ; C, CYBER 205 cu patru unități pipeline și operații triadice pe 64 biți.

Astfel, așa cum s-a afirmat anterior, r_∞ se dublează iar $n_{1/2}$ rămîne aproximativ nemodificat. Curbele B și C din fig. 2.26 ilustrează aceasta. De fapt $n_{1/2}$ este mărit cu aproximativ 50% datorită timpului de execuție a instrucțiunii de selectare a legăturii, ignorată în analiza de mai sus, dar care trebuie executată înaintea instrucțiunilor vectoriale care trebuie egate împreună. Creșterea lui $n_{1/2}$ este ilustrată în fig. 2.26 de curba C.

Tabelul 2.5 prezintă rezultatele măsurătorilor lui r_∞ și $n_{1/2}$, în comparație cu cele anterioare obținute la un CRAY X-MP cu un singur CPU. Performanța specifică π_0 măsoară performanța vectorului scurt (vezi §1.3.5), de aici se poate afirma imediat că performanța pentru vectorii mici

este întotdeauna mai bună la X-MP decît la CYBER 205, chiar pentru configurația cu 4 unități pipeline. Pe de altă parte — exceptînd 205 cu o unitate pipeline în modul pe 32 biți — performanța pentru vectori lungi, măsurată de r_∞, este întotdeauna la CYBER 205 mai mare decît la X-MP/1 Deci ,trebuie să fie o valoare a lungimii vectorului, să spunem n, pentru care CYBER 205 este mai rapid și sub care CRAY X-MP este mai rapid.

Valoarea lui $\hat{n}$ se poate obține prin egalarea performanțelor celor 2 mașini. Dacă folosim exponentul (2) pentru CYBER 205 și (1) pentru CRAY X-MP, se poate scrie

$$(\hat{n} + n_{1/2}^{(2)})/r_\infty^{(2)} = (\hat{n} + n_{1/2}^{(1)})/r_\infty^{(1)} \qquad (2.14a)$$

de unde

$$\hat{n} = n_{1/2}^{(2)}(1 - \gamma)(\alpha - 1) \qquad (2.14b)$$

unde $\alpha = r_\infty^{(2)}/r_\infty^{(1)}$ și $\gamma = \pi_0^{(2)}/\pi_0^{(1)}$ sînt raportul performanțelor asimptotice, respectiv specifice. S-a folosit ecuația (2.14b) pentru calculul valorilor lui n în tabelul (2.5).

Tabelul 2.5 **Performanța asimptotică r_∞ și $n_{1/2}$ și performanța specifică $\pi_0 = r_\infty/n_{1/2}$ pentru operații continue cu operanzi aflați în memorie, la CYBER 205 și CRAY X-MP cu un CPU. $\hat{n}$ este lungimea vectorului pentru care CYBER 205 atinge o performanță mai mare decît CRAY X-MP/1.**

Calculator	r_∞(Mflop/s)			$n_{1/2}$		π_0(M/s)		$\hat{n}$
	Biți	Diadă	trîadă	Diadă	Triadă	Diadă	Triadă	Diadă
CYBER 205	64	50	100	50	75	1	1,3	
un pipe	32	100	200	100	150	1	1,3	58
CYBER 205	64	100	200	100	150	1	1,3	58
două pipe	32	200	400	200	300	1	1,3	26
CYBER 205	64	200	400	200	300	1	1,3	26
4 pipe	32	400	800	400	600	1	1,3	20
CRAY X-MP/1	64	70	148	53	60	1,3	2,4	—

Parametrii de performanță din tabelele **2.4** și 2.5 sînt valabili numai dacă elementele succesive implicate sînt memorate în adrese succesive de memorie. Se spune despre acești vectori că sînt *contigui* și memoria tuturor calculatoarelor este astfel organizată încît accesul la astfel de vectori să se facă fără conflict. *Pasul* unui vector este intervalul dintre adresele de memorie ale elementelor succesive ale unui vector. Un vector contiguu este deci un vector cu pas unu și orice alt tip de vector este *ne-contiguu* În general, vectorii pot avea alte valori pentru pas. De exemplu, dacă elementele unei matrici (n×n) sînt memorate continuu coloană cu coloană (modul FORTRAN), liniile matricii formează vectori cu pasul constant de n. Se mai spune uneori că acești vectori sînt periodici. Alți vectori pot avea elemente a căror locație este specificată de o listă de adrese care poate avea valori arbitrare. Acești vectori sînt numiți aleatori și sînt accesați cu instrucțiuni de dispersare/grupare (sau adresare indirectă).

Deoarece calculatoarele sînt optimizate în mod obișnuit pentru accesul rapid la elementele unui vector contiguu, de obicei performanța lor se degradează (uneori dramatic) dacă nu se folosesc vectori de acest tip. Situația

este în special adevărată la CYBER 205 și, ca un exemplu, să considerăm timpul de execuție al operației X=Y*Z între vectori memorați aleatoriu. Deoarece la CYBER 205 există numai instrucțiuni vectoriale pentru vectori contigui, operația menționată trebuie executată în mai multe etape : în primul rînd cei doi vectori de intrare Y și Z trebuie „grupați" în doi vectori temporari contigui; se execută apoi operația, ce produce ca rezultat un vector contiguu temporar; rezultatul este dispersat în locațiile aleatoare ale vectorului Z. Se poate calcula timpul de execuție ,pe baza formulelor din tab. 2.4.

$$\begin{aligned} &\text{grupare } Y && =(69+1{,}25n)\tau \\ &\text{grupare } Z && =(69+1{,}25n)\tau \\ &\text{înmulțire contiguă} && =(52+0{,}5n)\tau \\ &\text{dispersare } X && =(83+1{,}25n)\tau \\ &\text{timp total} && =(273+4{,}25n)\tau \\ & && =(4{,}25/\tau^{-1})(n+64) \end{aligned} \quad (2.15)$$

Deoarece τ^{-1} corespunde la 50 Mflop/s, parametrii efectivi ce definesc operația cu vectori ne-contigui sînt:

$$r_\infty = \tau^{-1}/4{,}25 = 12 \text{ Mflop/s}; \; n_{1/2} = 64 \quad (2.16)$$

Astfel, am găsit că utilizarea vectorilor ne-contigui a degradat performanța printr-un factor de 10 față de performanța de 100 Mflop/s pentru vectori contigui. Deoarece timpul de execuție al operațiilor cu vectori necontigui este dominat de timpul operațiilor de dispersare/grupare care nu este micșorat prin introducerea unor unități pipeline suplimentare, performanța de 10 Mflop/s nu se va modifica dacă se crește numărul unităților pipeline.

Desigur, ar fi absurd să se programeze în întregime CYBER 205 cu operații vectoriale ne-contigui de tipul discutat în ultimul paragraf. În primul rînd, toate problemele trebuie structurate astfel ca numărul operațiilor cu vectori ne-contigui să se reducă la minim posibil, chiar la zero; și ,în al doilea rînd, dacă operațiile cu vectori ne-contigui sînt de inevitat, este de dorit să fie grupate astfel ca multe operații ne-contigui (în loc de una în exemplul de mai sus) să se execute cu vectorii temporar contigui. Astfel se amortizează efectul overhead-ului introdus de operațiile de dispersare/grupare, în cazul mai multor operații vectoriale. Chiar așa, performanța unei operații diadice cu vectori contigui și necontigui reprezintă cazul cel mai bun, respectiv cel mai rău, performanța unei anumite probleme aflîndu-se între cele 2 valori limită. Faptul că domeniul de variație al performanței este la CYBER 205 atît de mare indică că pentru a obține performanța cea mai bună poate fi necesară o restructurare considerabilă a programului.

2.3.7. ETA[10]

ETA[10] este primul produs al firmei ETA Systems Inc (1450 Energy Park Drive, St Paul, MN 55108, SUA), înființată de Control Data Corporation în august 1983 pentru a scurta ciclul de dezvoltare al noilor supercalculatoare și, în particular, pentru continuarea dezvoltării seriei CYBER

205. Acest calculator de 10 Gflop/s a fost anunţat în 1986, iar prima livrare s-a efectuat la începutul anului 1987. Sistemul urmează să fie transformat într-un calculator de 30 Gflop/s, ETA^{30}, prin 1992, posibil prin folosirea tehnologiei cu arseniură de galiu. S-au comandat calculatoare ETA^{10} pentru National Advanced Scientific Computing Centers la Princeton, Minnesota şi Florida State Universities, ca şi pentru Supercomputer Applications Laboratory (SAL) de la University of Georgia, Athens. În exteriorul SUA, s-au făcut comenzi pentru German Weather Service din Offenbach, R.F.G. şi pentru Atmospheric and Environmental Service of Canada. Prima livrare s-a făcut în ianuarie 1987 Universităţii de stat din Florida, la Tallahasse.

În fig. 2.27 (a) se prezintă arhitectura de ansamblu a calculatorului ETA^{10}, cu 2, 4, 6 sau 8 CPU şi de la 2 la 18 unităţi de I/E, ce lucrează cu o memorie comună partajată de 64, 128, 192 sau 256 Mw (64 biţi).

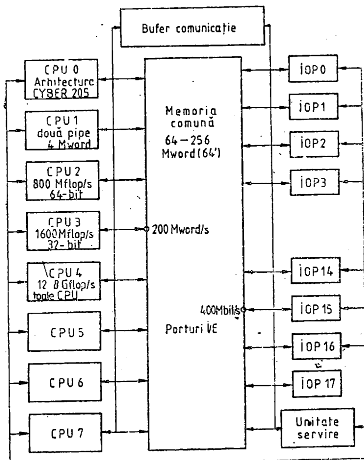


Fig. 2.27 (a) Diagrama bloc a calculatorului ETA^{10}.

Fiecare CPU are aceeaşi arhitectură cu un CYBER 205 cu 2 unităţi pipeline şi 4 Mw (64 biţi) de memorie locală, aşa cum s-a prezentat în §2.3.1 la §2.3.5 În fig. 2.27 (b) se prezintă instalarea de la Tallahasse. Cele 2

cabinete din faţă au fiecare cîte 4 CPU cu memoria lor locală, în timp ce în cabinetul mai înalt din spatele lor se află memoria comună şi unităţile de I/E. Deşi memoria este organizată ierarhic, programatorii vor utiliza un spaţiu uniform de adrese virtuale, adresate cu 48 biţi, ca la CYBER 205. Versiunea ETA[30] va avea 16 Mw de memorie locală pe procesor şi 1 Gw de memorie comună. Performanţa maximă a fiecărui CPU ETA[10] este pentru operaţii triadice de 800 Mflop/s în modul pe 64 biţi şi de 1600 Mflop/s în modul pe 32 biţi, ce corespund la un ceas cu perioadă de 5 ns. Este o creştere de 4 ori faţă de CYBER 205 unde perioada ceasului era

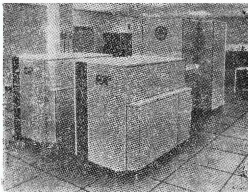
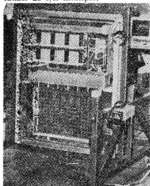


Fig. 2.27 cont. (b) O instalare a unui ETA[10] la Florida State University, Tallahasse, efectuată în ianuarie 1987. Fiecare din cele două cabinete din plan apropiat conţine cîte patru CPU, cu 4Mw de memorie locală fiecare. Memoria comună de 256Mw şi unităţile de I/E se află în cabinetul mai înalt din spate.

de 20 ns. Performanţa maximă a sistemului complet cu 8 CPU este, prin urmare, de 12,8 Gflop/s în modul pe 32 biţi, atingîndu-se astfel scopul propus de firmă. Primele maşini vor avea o perioadă a ceasului de 7 ns.

Fiecare CPU este conectat la memoria partajată printr-un port de mare viteză cu o lărgime de bandă de un cuvînt de 64 biţi pe perioada de ceas (200 Mw/s sau 12,8 Mb/s). Aceasta înseamnă 1/6 din viteza de transfer necesară alimentării celor 2 unităţi aritmetice pipeline ale unui CPU direct din memoria comună, de aceea se presupune că un număr substanţial de calcule se execută în interiorul CPU folosind memoria locală, înainte ca rezultatele să fie returnate memoriei partajate. Bufferul de comunicare este o zonă de memorie de 1 milion de cuvinte, folosită pentru comunicarea şi sincronizarea activităţii multiprocesorului. Operaţiile de I/E se execută prin 18 porturi de I/E mai lente, cu lărgime de bandă de 400 Mb/s. Fiecare port de I/E conţine pînă la o 8 unităţi funcţionale, fiecare

cu un microprocesor 68020 pentru controlul perifericelor standard (unități de disc și bandă și rețelelor (Ethernet, Hyperchannel și LCNS) via canale de I/E multiple.



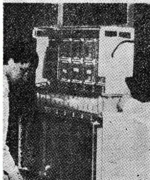


Fig. 2.27 cont. (c) (Sus) O placă CPU de la ETA[10], care conține toate circuitele a două unități pipeline de la CYBER 205. Deasupra acesteia este memoria locală 4Mw. (Dedesubt) Placa CPU plasată în hidrogen lichid. În timp ce placa CPU este scufundată complet în lichidul de răcire, memoria locală este răcită cu aer.

Cum viteza internă a dispozitivelor logice a crescut în interiorul capsulelor, importanța relativă a întîrzierii determinată de interconectări, înregistrată la trecerea semnalelor logice între capsule, a crescut. De aceea, un obiectiv important al proiectanților de supercalculatoare este reducerea acestei întîrzieri prin plasarea a cît mai multor porți logice pe fiecare capsulă reducînd numărul lor și, implicit, pe cel al interconexiunilor. Din păcate, creșterea nivelului de integrare VLSI conduce la folosirea tehnologiilor mai lente, inacceptabile în mod normal într-un calculator de mare performanță; de exemplu, se poate atinge o densitate mult mai mare de porți cu CMOS, în locul tehnologiei mai rapide ECL. ETA Systems a rezolvat această dilemă prin adoptarea tehnologiei mai lente CMOS dar cu densitate mai mare și a recîștigat viteza pierdută prin funcționarea capsulelor la temperatura hidrogenului lichid.

Principala inovație la calculatorul ETA[10] este tehnologică, deoarece arhitectura CPU este cunoscută încă de la mijlocul anilor' 60, la CDC STAR 100. ETA Systems a adoptat masive de porți CMOS cu densitate de 20 000 porți pe 1 cm pătrat. Aceste circuite produse de Honneywell pentru programul VHSIC (very high-speed integrated circuit) al Departamentului apărării, folosesc tehnologie de 1,25 micrometri. Datorită complexității, s-au încorporat scheme de auto-test în interiorul capsulelor. Tehnologia consumă energie numai la modificarea stării. În consecință

multi-capsulă (MCC), care pot fi folosite de pînă la 121 ansambluri organizate matricial 11×11, ca în fig. 2.30(a). MCC este o placă cu 14 straturi cu suprafaţa de 31×31 cm². Se montează împreună 13 astfel de plăci într-un ansamblu denumit stivă (vezi fig. 2.30(b)). Memoria principală a

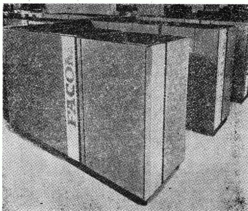


Fig. 2.28. Un calculator Fujitsu VP-200. Cele trei cabinete din dreapta sînt pentru memorie, unitatea scalară şi procesoarele de canal; cele două din stînga conţin unitatea vectorială şi memorie.

calculatorului VP-100/200 este construită cu circuite MOS static de 64 Kb, cu un timp de acces de 55 ns. Aceste circuite nu necesită aripioare pentru răcire şi sînt montate pe plăci cu 6 straturi. Fiecare placă conţine un masiv 4×32 de circuite de 64 Kb, rezultînd 1 MB.

În fig. 2.31 se prezintă arhitectura de ansamblu a procesoarelor vectoriale FACOM. Arhitectura unităţii vectoriale este ca la CRAY, în sensul că unităţi funcţionale multiple (pentru adunare, înmulţire şi împărţire în virgulă mobilă) lucrează cu o memorie constituită din registre vectoriale (64KB). Există o unitate scalară separată, ca la CYBER 205, cu o memorie tampon de 64 KB (acces în 5,5 ns). Memoria principală (acces în 55 ns) de 256 MB este organizată în 256 blocuri şi este conectată la registrele vectoriale prin două unităţi pipeline de încărcare/memorare. Cifrele anterioare sînt pentru VP-200; la VP-100 dimensiunea memoriei principale şi a registrelor vectoriale se înjumătăţeşte. Mai există 256 registre de mascare, fiecare pe 32 biţi (16 biţi la VP-100), folosite pentru memorarea vectorilor de mascare care controlează operaţiile vectoriale condiţionate şi operaţiile de modificare a vectorilor.

O caracteristică unică este posibilitatea reconfigurării dinamice a registrelor vectoriale fie ca 256 registre vectoriale pentru 32 elemente a 64 biţi, fie ca 128 registre a 64 elemente, . . ., etc. sau ca 8 registre a 1024 elemente. Lungimea registrelor vectoriale este menţionată de un registru special şi poate fi modificată cu o instrucţiune.

Perioada ceasului este la unitatea scalară de 15 ns, şi este denumită ciclul major. Unitatea vectorială lucrează cu o perioadă de 7,5 ns, ciclul

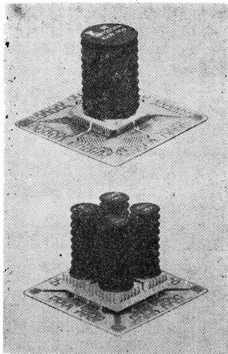
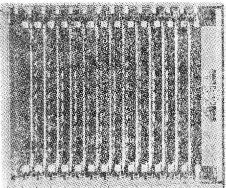


Fig. 2.29 Capsulele răcite cu aer ale procesoarelor vectoriale Fujitsu, (a) un circuit LSI ECL cu 400 porţi. (b) Patru circuite de 1 Kb de memorie montate pe un modul.



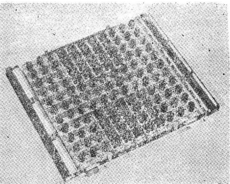


Fig. 2.30 Tehnologia folosită de procesoarele vectoriale Fujitsu. (a) O placă cu posibilitatea de a avea 121 circuite LSI. (b) O stivă de 13 plăci montate orizontal.

minor. La VP-200, unităţile pipeline de adunare şi înmulţire în virgulă mobilă pot produce două rezultate pe 64 biţi la fiecare perioadă de ceas, ceea ce înseamnă o performanţă maximă de 267 Mflop/s pentru operaţii diadice cu operanzii în registre şi o unitate pipeline, sau 533 Mflop/s pentru operaţii triadice cu operanzii în registre, care folosesc ambele unităţi pipeline simultan. La VP-100 aceste valori se înjumătăţesc. Unitatea pipeline pentru împărţire este mai lentă şi lucrează la 38 Mflop/s. La VP-200 fiecare unitate pipeline pentru încărcare/memorare poate produce 4 cuvinte pe 64 biţi la fiecare 15 ns, echivalentul unei lărgimi de bandă de 267 Mw/s (133 Mw/s la VP-200). Aceste valori reprezintă 2/3 din lărgimea de bandă necesară pentru o operaţie diadică cu argumentele şi rezultatele aflate în memoria principală. Astfel, spre deosebire de CRAY X-MP şi CYBER 205, Fujitsu VP nu are o lărgime de bandă suficientă pentru a executa astfel de operaţii cu memoria. De aceea, compilatorul are o sarcină dificilă, de a maximiza utilizarea registrelor vectoriale pentru rezultatele intermediare cu scopul limitării transferurilor cu memoria principală.

Setul de instrucţiuni de la Fujitsu VP este identic cu cel de la IBM 370, la care se adaugă instrucţiuni vectoriale; într-adevăr modulele de cod generate pe IBM 370 se pot executa fără modificări pe VP. Instrucţiunile vectoriale includ evaluarea condiţionată a operaţiilor aritmetice vectoriale controlate cu o mască cu un bit pentru fiecare element al vectorului (ca la CYBER 205, §2.3.4); comprimarea şi extensia vectorilor conform unei condiţii ;adresare vectorială indirectă, adică o instrucţiune de dispersare grupare ca cea descrisă la CRAY X-MP în §2.2.4. Această instrucţiune poate grupa 4 elemente la fiecare 15 ns.

Se anticipează că cei mai mulţi utilizatori îşi vor scrie programele în FORTRAN, de aceea se dezvoltă foarte mult soft interactiv pentru optimizarea şi vectorizarea unor astfel de programe (Kamiya, Isobe, Takashima şi Takiuchi 1983, Matsuura, Miura şi Makino 1985). De exemplu, vectorizarea instrucţiunilor IF pune o problemă specială, iar compilatorul vectorial FORTRAN 77/VP selectează pe cea mai bună din cele trei metode disponibile. Acestea sînt: (a) evaluarea condiţionată prin folosirea unor măşti; (b) selectarea elementelor care participă la operaţie într-un vector comprimat; şi (c) folosirea adresării vectoriale indirecte pentru selecţia elementelor. Compilatorul compară cele trei metode pe baza frecvenţei relative a operaţiilor de încărcare/memorare în buclele DO şi a fracţiei din numărul elementelor vectorului care participă la execuţia operaţiei (raportul real). Dacă acest raport este mediu spre mare, cel mai bine este să se execute o operaţie aritmetică mascată; în celelalte cazuri, metoda comprimării este cea mai bună, cînd frecvenţa operaţiilor de încărcare/memorare este mică, iar adresarea indirectă cînd frecvenţa este mare. Interactivitatea ia forma unor sugestii pentru programator asupra modului cum să restructureze programul pentru a creşte nivelul vectorizării.

2.4.2. Hitachi HITAC S-810

Hitachi HITAC S-810 a fost al doilea procesor vectorial japonez produs, prima livrare efectuîndu-se Universităţii Tokyo în 1984. Alte maşini au fost instalate pentru uzul intern al companiei. Calculatoarele

Hitachi S-810 model 10 şi model 20 sînt similare, în ceea ce priveşte arhitectura de ansamblu, maşinilor Fujitsu, aşa cum se poate observa prin compararea fig. 2.31 şi 2.32. Diferenţa principală constă în aceea că S-810

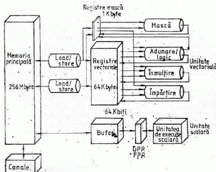


Fig. 2.31 Diagrama bloc a procesorului vectorial Fujitsu. (FPR notează registrele pentru numere în virgulă mobilă, iar GPR registrele de uz general).

are mai multe unităţi pipeline (Nagashima, Inagami, Odaka şi Kawabe 1984). La S-810 există trei unităţi pipeline de încărcare/memorare, în comparaţie cu numai două la VP. Aceasta înseamnă că operaţiile diadice cu operanzi din memorie pot fi executate de S-810 la întreaga viteză. Memoria principală are capacitatea de 256 MB (timp de acces 40 ns) şi mai există 64 KB de registre vectoriale. Ca şi la Fujitsu VP, mediul format din registre poate fi reconfigurat dinamic pentru memorarea unor vectori de lungimi diferite.

Modelul 20 are 12 unităţi pipeline aritmetice în virgulă mobilă (4 pentru adunare, 2 pentru înmulţire/împărţire urmate de adunare, şi 2 pentru înmulţire urmată de adunare). La ambele modele perioada ceasului este de 14 ns, care corespunde unei performanţe maxime de 71,4 Mflop/s pe pipeline pentru operaţii cu operanzi în vectori, cu un total de 857 Mflop/s pentru cele 12 unităţi pipeline. Totuşi, dacă se ia în considerare timpul necesar încărcării registrelor vectoriale se obţine o valoare reală de 60 Mflop/s, dacă se folosesc toate unităţile pipeline. Sistemul este optimizat pentru evaluarea expresiilor de tipul A=(B+C)*D care necesită trei încărcări vectoriale şi o memorare vectorială şi, deci, folosesc toate cele trei unităţi de încărcare/memorare. Modelul 10 are 6 unităţi pipeline, aritmetice, spaţiul de memorare format din registre, ca şi memoria principală reduse la jumătate. Performanţa lui maximă este de 315 Mflop/s. Ca la toate calculatoarele prezentate, performanţa reală va fi mai mică decît cea maximă, datorită problemelor puse de accesul la memorie. S-a

executat testul (r_∞, $n_{1/2}$), prezentat în §1.3.3., pentru un număr de bucle DO vectorizate (instrucţiunea 10 din programul (1.5). În tab. 2.6 se dau rezultatele obţinute pe S-810 modelele 10 şi 20 cu reprezentare pe 32 şi

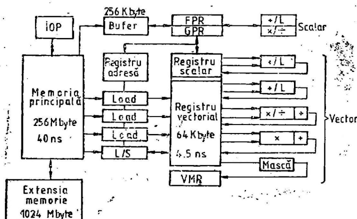


Fig. 2.32 Diagrama bloc a calculatorului Hitachi S-810 model 10.
(VMR notează registrul de mascare vectorială, iar L operaţii logice.)

64 biţi. Se observă pentru modelul 10 o performanţă maximă de aproximativ 240 Mflop/s pentru cazul a 4 operaţii, care este o expresie ce foloseşte integral sistemul hardware. În acest caz, există trei vectori de intrare şi un vector de ieşire care ocupă cele 4 unităţi pipeline de acces la memorie. În plus expresia foloseşte cele două unităţi pipeline pentru adunare şi cele două pentru înmulţire. Deoarece unităţile pipeline au perioada ceasului de 14 ns, s-ar obţine 71 Mflop/s pe pipeline, de unde un maxim de 284 Mflop/s. Valoarea măsurată de 240 Mflop/s este mai mică datorită timpului necesar încărcării registrelor vectoriale.

Dacă sînt mai mult de trei vectori de intrare şi unul de ieşire, lărgimea de bandă a memoriei este insuficientă pentru alimentarea unităţilor pipeline cu date la viteza cerută. Performanţa se degradează şi, apoi expresia

Tabelul 2.6. **Rezultatele testului (r_∞, $n_{1/2}$) pentru Hitachi S-810/10, iar pentru modelul 20 în paranteze. Variabilele notate cu litere mari sînt vectori, celelalte scalari. (Date puse la dispoziţie de M. Yasumura, Hitachi Central Research laboratory, Tokyo).**

Operaţie : instrucţiunea 10 program (1.5)	Pas	Precizie biţi	r (Mflop/s)	$n_{1/2}$
Diadă	1	32	60(97)	60(130)
		64	62(119)	73(143)
A = B + C	8	32	31(61)	46(108)
		64	56(110)	92(208)
Triadă	1	32	118(180)	71(126)
		64	121(238)	73(152)
A = B + e × C	8	32	43(85)	27(61)
		64	66(131)	41(108)
4 operaţii	1	32	238(345)	91(157)
		64	231(489)	88(190)
A = B + (exC + f × D)	8	32	85(163)	24(58)
		64	134(263)	49(111)

trebuie tratată ca o combinaţie de operaţii diadice sau triadice mai simple, a căror performanţă o vom trata în continuare. Expresii mai simple ca cele diadice sau triadice nu pot folosi integral unităţile aritmetice pipeline şi observăm viteze de aproximativ 60 şi 120 Mflop/s respectiv, pentru vectori contigui cu un pas de o unitate. Această valoare se poate degrada la jumătate sau o treime datorită conflictelor de acces la blocurile de memorie dacă vectorii nu sînt contigui, în cazul nostru pentru un pas de 8. În concluzie, se poate spune că, pentru vectori contigui, $r_\infty = 60$ Mflop/s pe operaţie aritmetică în expresie (pînă la un maxim de 4), cu o degradare de pînă la un factor de 3 pentru vectori memoraţi impropriu. Aceste rezultate nu par a fi afectate de precizia operaţiilor aritmetice. Astfel, la modelul 10 performanţa se află în domeniul de la 30 la 240 Mflop/s, funcţie de circumstanţe. Valorile lui $n_{1/2}$ se află în domeniul de la 40 la 90.

Rezultatele corespunzătoare modelului 20 apar în tab. 2.6 în paranteze. Acest model are un număr dublu de unităţi pipeline şi observăm că r_∞ şi $n_{1/2}$ au şi ele valori aproape duble, conducînd la o performanţă de între aproximativ 60 şi 480 Mflop/s funcţie de circumstanţe. Se poate face comparaţie cu valoarea de 630 Mflop/s furnizată de producător. Deoarece şi $n_{1/2}$ este dublu, sînt necesari vectori de două ori mai lungi pentru a se obţine aceeaşi fracţie din performanţa maximă.

Hitachi S-810 este răcit cu aer, ca şi Fujitsu VP şi foloseşte tehnologie LSI ECL cu 550 porţi pe capsulă (timp de propagare 350 ps), sau 1500 porţi pe capsulă (timp de propagare 450 ps). Registrele vectorilor folosesc circuite bipolare de 1 Kb cu timp de acces de 4,5 ns, iar memoria principală foloseşte circuite RAM static CMOS de 16 Kb. Fiecare placă poate avea 40 capsule şi are 14 straturi.

Ca şi la Fujitsu VP, setul de instrucţiuni este o extensie a celui de la IBM 370, iar pentru IBM FORTRAN 77 este prevăzut un compilator vectorizant. Tehnicile folosite pentru vectorizare sînt descrise de Yasamura et al (1984). Modulele obiect generate pe IBM pot fi executate fără nici o modificare.

2.4.3. NEC SX1/SX2

Calculatoarele SX1 şi SX2 ale firmei Nippon Electric Corporation au apărut ultimele, dar SX2 are performanţa teoretică cea mai mare, de peste 1 Gflop/s (fig. 2.33). În anul 1985 s-au livrat primele două SX2 Universităţii Osaka şi firmei Sumito Trading Company. În Europa sînt vîndute de Mitsui şi Company Europe Group.

Calcuatoarele SX folosesc porţi bipolare (CML) cu densitatea de 1000 pe capsulă şi timp de propagare pe poartă de 250 ps. Memoria cache şi registrele vectoriale sînt construite cu circuite RAM bipolare de 1 Kb cu timp de acces de 3,5 ns. Aceste circuite (36 din ele) sînt încapsulate pe un suport ceramic de 10 cm² (fig. 2.34(a)), care este scufundat într-un modul de răcire (fig. 2.34(b)), prin care circulă apă. Memoria principală de 256 MB se compune din circuite RAM static MOS de 64 Kb cu timp de acces de 40 ns. Memoria extinsă de 2 GB se compune din circuite MOS dinamic.

Arhitectura calculatorului NEC SX2 este prezentată în fig. 2.35 (Watanabe 1984). Există patru unități pipeline vectoriale generale, fiecare calculînd al patrulea element al unei operații vectoriale. Fiecare este o

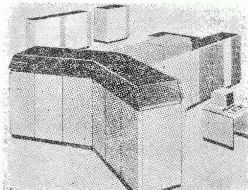


Fig. 2.33 Calculatorul Nippon Electric Company S×2

combinație între un pipeline pentru înmulțire/împărțire și unul pentru adunare. Astfel, elementele unei operații vectoriale sînt distribuite unităților pipeline disponibile, ca la CYBER 205. Perioada ceasului este de 6 ns, astfel că atunci cînd lucrează simultan toate cele 8 unități pipeline în virgulă mobilă se atinge un maxim $r_\infty = 1333$ Mflop/s. Aceste cifre se referă la modelul SX2. Calculatorul SX1 are perioada ceasului de 7 ns și jumătate din numărul de unități pipeline. $r_\infty = 570$ Mflop/s, maximum. Viteza de transfer între memorie și registrele vectoriale este de 8 numere într-o perioadă de ceas.

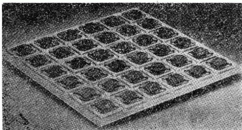
Limbajul preferat este FORTRAN 77. Există un vectorizator, analizor și optimizator automat care asistă restructurarea codului FORTRAN pentru a se obține un nivel mai bun de vectorizare. Spre deosebire de celelalte mașini, setul de instrucțiuni nu este compatibil cu cel de la IBM 370, deci modulele obiect IBM nu vor putea fi executate pe această mașină.

2.4.4. Compararea performanțelor

S-au efectuat un număr substanțial de teste pentru a compara calculatoarele vectoriale descrise, iar unele rezultate apar în tab. 2.7. Primele trei linii dau performanța medie în Mflop/s pentru trei din așa numitele *bucle Livermore* (McMahon 1972, Arnold 1982, Riganati și Schneck 1984).

Lawrence Livermore Laboratory a selectat 14 astfel de bucle, ca fiind tipice pentru activitatea sa. Noi am selectat trei din ele, care ilustrează atît performanța cea mai bună, cît și cea mai slabă a calculatorului. Bucla 3, produsul interior, se află la baza celor mai multe rutine de algebră liniară. Cele mai multe calculatoare vectoriale optimizează execuția acestei bucle, pentru care se obțin, de obicei, performanțele cele mai mari. Performanțele corespunzătoare buclelor 6 și 14 sînt caracteristice calculelor scalare deoarece buclele DO implică recurențe, iar posibilitatea de vectorizare este redusă sau eliminată. Aceste probleme sînt rezolvarea unui sistem tridiagonal de ecuații și simularea deplasării particulelor în plasmă.

Buclele Livermore sînt numite teste nucleu (kernel benchmarks) deoarece constau din segmente mici de program, mai degrabă decît din programe de rezolvare completă a unei probleme. Observăm prin compararea buclei Livermore 3 cu buclele 6 sau 14 că performanța variază cu un factor de cel puțin 10. De aceea, este interesant de văzut ce performanță se găsește la o problemă completă care ar implica atît bucle vectori-



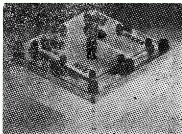


Fig. 2.34 Tehnologia de răcire cu apă folosită de NEC SX1/SX2. (a) O capsulă de 10 cm conține 36 circuite LSI, fiecare a 1000 porti logice. (b) Modulul de răcire cu lichid.

zabile cît şi nevectorizabile, împreună cu un cod scalar. Pentru aceasta Dongarra (1985) a interpretat o serie de rezultate ale evaluării bine-cunoscutelor rutine *LINPACK* pentru rezolvarea a 100 ecuaţii liniare prin decompoziţie triunghiulară (LU) şi substituire (Dongarra et al.

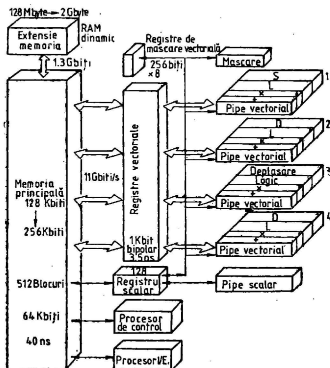


Fig. 2.35 Diagrama bloc a calculatorului NEC S×2

1979). Liniile 4 şi 5 din tab.2.7 dau performanţele codului FORTRAN şi a codului special optimizat în limbaj de asamblare pentru aceeaşi problemă. Rezolvarea a 100 ecuaţii liniare nu este o problemă suficient de complexă pentru a demonstra posibilităţile unui supercalculator, în particular unul cu mai multe CPU. De aceea, în linia 6 se arată că cea mai bună performanţă se obţine la rezolvarea a 300 ecuaţii liniare, cu metoda matrice-vector (vezi paragraful următor). Performanţa obţinută pentru acest program de test indică ce s-ar putea obţine cu un cod optimizat cu atenţie.

Analiza tab.2.7 arată că nu este posibilă o distincţie clară între performanţele caluculatoarelor. Toate calculatoarele au performanţe similare şi ne putem aştepta să lucreze la cîteva zeci de Mflop/s cu un cod FORTRAN ne-optimizat şi cîteva sute de Mflop/s pentru un cod optimizat cu atenţie (posibil în limbaj de asamblare). Care calculator are cea mai bună performanţă pentru o problemă particulară, probabil depinde de atenţia cu care este optimizat programul. De exemplu, se poate obţine o îmbunătăţire substanţială la CRAY X–MP dacă decompoziţia LU se exprimă în termeni de operaţie matrice-vector, în locul operaţiilor vector-vector

(vezi linia 6). O operaţie matrice-vector poate fi definită în modul multitasking foarte eficient cu ajutorul mai multor CPU (Dongarra şi Eisenstat 1984, Chen et al 1984). La CRAY X—MP se obţine cea mai bună performanţă dacă se elimină simultan trei linii (Dongarra şi Hewitt 1985). În acest caz se obţin 718 Mflop/s la rezolvarea a 1000 ecuaţii cu un CRAY X—MP cu 4 CPU. Alte detalii privind compararea performanţelor supercalculatoarelor prezentate apar în lucrările lui Bucher (1984), Lubeck et al (1985) şi Bucher şi Simmons (1986).

Tabelul 2.7 **Cîteva rezultate comparative ale execuţiei unor programe de test de către calculatoare vectoriale pipeline. Pefroamanţa medie este dată în Mflop/s, pe 64 biţi.**

Problema	CRAY X—MP[d]	CYBER 205[f]	IBM[f] 3090VF[d]	Fujitsu VP200 VP400	Hitachi S-810/20	NEC SX2	CRAY-2[d]	ETA[10]:
Performanţa	210(1)	100(1)	108(1)	533(2)	840	1300	488(1)	1250(1)
teoretică	420(2)	200(2)	216(2)	1142(4)			976(2)	5000(4)
maximă	840(4)	400(4)	432(4)				1951(4)	10000(8)
$(r_\infty, n_{1/2})$	—	—	—	—	(119,143)	—	(56,83)(1)	—
FORTRAN	(§1.3.3)							
Livermore 3	96(1)[e]	86(1)[c]	—	331(2)[e]	212[e]	561[g]	63(1)	—
produsul interior				359(4)[g]	332[i]		91(1)	
Livermore 6	8(1)	5(1)[c]	—	10(2)[e]	5[c]	13[g]	4(1)	—
tridiagonal				10(4)[g]	32[i]	—	8(1)	—
Livermore 14	7(1)[e]	5(1)[c]	—	14(2)[e]	9[e]	24[g]	6(1)	—
analiza particulelor				15(4)[g]	11[l]	—	9(1)	—
LINPACK[a]	24(1)	20(2)	(12(1)	18(2)[k]	17	46[j]	15(1)[k]	—
FORTRAN[m]								
n = 100								
Asamblor[b]	44(1)	25(2)	—	—	—	—	—	—
bucla interioară								
n = 100								
Matrice-vector[c]	171(1)	31(2)	18(1)[h]	183(2)	158[h]		309	93(1)
cel mai bun	257(2)		27(1)	230(2)[h]				
asamblor	480(4)							
n = 300								

Note

a Rezolvarea ecuaţiilor liniare cu DGEFA şi DGESL pentru matrici de ordinul n (Dongarra et al 1979).

b Subrutine de bază pentru algebra liniară (BLAS), optimizate în limbaj de asamblare (Dongarra 1985).

c Metoda matrice-vector a lui Dongarra şi Einsenstat (1984). Matrice de ordinul 300.Rezultatul cel mai bun în limbaj de asamblare (Dongarra 1985).

c Numărul de CPU folosit apare în paranteză.

e Fuss şi Hollenberg 1984.

f Numărul de unităţi pipeline apare în paranteză.

g Van der Steen 1986.

h Toate în FORTRAN

i După optimizare (Nagashima et al 1984).

j Dongarra 1986

k Dongarra şi Sorensen 1987.

l Un prototip CRAY-2 cu un procesor (Bruijnes 1985).

m Cod FORTRAN cu bucla BLAS (Lawson et al 1979, Dongarra 1985). Directivele FORTRAN sînt permise.

n Numărul din paranteză indică modelul.

2.5 FPS AP-120B și derivatele sale

FPS AP—120 B și derivatele sale — AP 190 L, FPS — 100, 164, 164/MAX, 264 și FPS 5000 — sînt membrii unei singure familii de calculatoare ce se bazează pe o arhitectură comună, cea a calculatorului FPS AP—120B. Aceste calculatoare au fost redenumite astfel: versiunea MSI originală a lui FPS—164 (acum nu mai este produs) este denumită M140, ultima versiune VLSI (la început FPS—364) se cheamă M30, FPS—164/MAX este denumit M145, iar FPS—264 este acum M60. Toate sînt produse de Floating Point Systems Inc. (PO BOX 2 489 Portland, Oregon 97223, S.U.A.), în Beaverton lîngă Portland. Compania a fost fondată în 1970 de C.N.Winningstad pentru a produce unități în virgulă mobilă de mare performanță, dar cu un cost scăzut, pentru a crește performanțele minicalculatoarelor, în particular la aplicațiile de prelucrare a semnalelor. Începînd cu anul 1971, firma a produs unități în virgulă mobilă pentru a fi incluse în alte mașini (de exemplu, Data General). Prima mașină vîndută, AP—120B, a fost proiectată de George O'Leary și Alan Chalesworth și are o performanță de 12 Mflop/s. Livrările au început în 1976, iar la sfîrșitul anului 1985 erau instalate aproximativ 4.400 mașini. FPS100 este o versiune mai ieftină a lui AP—120B. AP—120B a fost proiectat pentru a fi atașat ca extensie la minicalculatoare, iar o versiune cu o memorie mai mare, AP—190L, a fost proiectată pentru calculatoare mari ca cele din seria IBM 370.

În anul 1980 s-a extins utilizarea calculatorului AP—120B de la aplicații de prelucrare a semnalelor la calcule științifice generale, prin creșterea lungimii cuvîntului de la 38 biți la 64, iar a adresei de la 16 la 24 biți. Capacitatea memoriei a fost sporită, la început la 1 Mw, apoi la 7,25 Mw. Noua mașină obținută, FPS—164, a început să fie comercializată în 1981. Pînă în 1985 s-au vîndut 180 FPS—164. Deși capabil să rezolve probleme mult mai complexe decît AP—120B, AP—164 nu execută mai rapid operațiile aritmetice—într-adevăr performanța sa maximă de 11 Mflop/s este mai mică decît cea a lui AP—120B cu un Mflop/s. În 1984 s-a obținut prima creștere a vitezei de calcul aritmetic prin introducerea unei plăci denumită accelerator matricial (MAX). Fiecare placă MAX poate fi considerată din punct de vedere computațional ca echivalentul a două CPU FPS—164, astfel că o mașină cu aproximativ 15 plăci are o performanță teoretică maximă de 41 Mflop/s, corespunzătoare a 31 CPU FPS—164. Atît AP—120B, cît și FPS—164 sînt implementate în tehnologie TTL. Următoarea îmbunătățire a performanței s-a realizat în 1985, cînd s-a anunțat versiunea în tehnologie ECL a mașinii FPS—164, denumită FPS—264, cu o performanță maximă de 38 Mflop/s. FPS—164 a fost proiectat în 1979 în tehnologie MSI cu între 10 și 100 porți pe circuit. Mașina a fost reproiectată în tehnologie VLSI CMOS și este vîndută sub numele FPS M 30, avînd posibilitatea de a fi conectată la microVAX și stații de lucru SUN. Arhitectura este aceeași ca la FPS—164.

FPS—164/MAX este un calculator SIMD deoarece cele 31 CPU ale unei configurații complete lucrează sincron (lock-step) ca urmare a execuției unui flux unic de instrucțiuni. O altă dezvoltare a arhitecturii

AP—120B s-a realizat în direcţia MIMD. Seria de calculatoare FPS—5000, anunţată în 1983, are un procesor de control, pînă la trei co-procesoare aritmetice şi un procesor de I/E care comunică printr-o magistrală comună cu memoria comună şi un calculator gazdă. Fiecare co-procesor aritmetic (AC) are propria unitate de control, şi poate executa o rutină proprie, diferită de ceea ce execută celelalte AC. Conform clasificării din 1.2.6., FPS—5000 este un calculator MIMD cu memorie partajată, conectat prin magistrală. Co-procesorul aritmetic XP—32 este un proiect nou datorat lui Pincus § Kallio, dar respectă principiile generale ale calculatorului AP—120B.

Toate calculatoarele prezentate mai sus sînt denumite masive de procesoare deoarece au fost proiectate să prelucreze eficient masive de numere. Arhitectural, toate sînt calculatoare pipeline, avînd un număr mic de unităţi aritmetice pipeline ce lucrează cu o memorie şi registre comune. Din acest punct de vedere, au o arhitectură similară cu CRAY—1. Este important să înţelegem că deşi aceste calculatoare sînt denumite masive de procesoare, ele nu sînt calculatoare de acest tip, ca ICL sau DAP(vezi capitolul 3). Ambiguitatea semnificaţiei expresiei „masiv de procesoare" ne-a determinat să evităm utilizarea ei în această carte. Totuşi, termenul este folosit în mod obişnuit pentru calculatoarele descrise aici, în special de către producători. Unii preferă să interpreteze iniţialele AP ca *procesoare ataşate* (attached processors), deoarece cele mai multe au nevoie de un calculator gazdă.

Vom începe prin descrierea detailată a părintelui familiei, AP—120B (§ 2.5.1 la § 2.5.6), continuăm cu aspecte speciale privind calculatoarele FPS—164 şi 264 (§ 2.5.7), FPS 164/MAX (§ 2.5.8) şi FPS—5000 (§2.5.10). În § 2.5.9 se discută utilizarea mai multor FPS—164 ataşate la o gazdă pentru a forma un masiv de procesoare slab interconectate (LCAP).

2.5.1. FPS AP-120B

În afara documentaţiei de firmă, referinţele principale care descriu AP-120B au ca autori pe Wittmayer (1978), Harte (1979) şi Charlesworth (1981).

Fig. 2.36 prezintă o instalare a unui calculator gazdă PDP 11/34, aflat în partea inferioară a rack-ului, un AP-120B, două unităţi de disc, şi o unitate de bandă magnetică şi o imprimantă. Consola nu apare în imagine. Maşina consumă mai puţin de 1,3 KW (în comparaţie cu aproximativ 115 KW la CRAY X-MP). Răcirea se face cu aer forţat. Maşina poate fi transportată de un singur om şi conectată la o priză standard de 13 A. Spre deosebire de CRAY X-MP şi CYBER 205, nu sînt necesare instalaţii de răcire sau generatoare auxiliare.

Fig. 2.37 prezintă o imagine a calculatorului AP-120B cu o placă logică parţial extrasă. În partea superioară pot fi văzute ventilatoarele care suflă aerul pe direcţie verticală. Şasiul are capacitatea de 28 plăci care pot fi alese pentru a satisface condiţiile particulare pentru memorie şi intrare/ieşire. În fig. 2.38 apar două plăci în detaliu. Fiecare placă are 6 straturi, 3 pentru semnale logice şi 3 pentru tensiunile de alimentare de 5 V şi +12V. Sursele de alimentare sînt montate separat şi nu apar în figură.

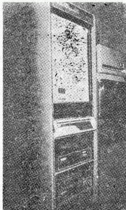


Fig. 2.36 O instalare FPS AP-20120B. Calculatorul AP-120B este conectat la un PDP 11/34, care are două unități de disc, o unitate de bandă și o imprimantă. (Fotografie furnizată de Dl. Head și Floating Point Systems, SA Ltd.)

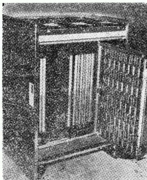


Fig. 2.37 Imagine ce prezintă plăcile calculatorului FPS AP-120B, modul de montare și răcirea cu aer. (Fotografie furnizată de Dl. Head și Floating Point Systems SA Ltd.)

2.5.2. Arhitectura

Arhitectura calculatorului AP-120B apare în fig. 2.39. Se bazează pe memorii multiple cu funcție specială, care alimentează două unități aritmetice pipeline via căi de date multiple. Mașina este condusă sincron pe baza unui singur ceas cu perioadă de 167 ns. Aceasta înseamnă că starea mașinii după o secvență de operații este întotdeauna cunoscută și reproductibilă. Modul de lucru al mașinii poate fi simulat exact, perioadă după perioadă, mașina nesuferind de incertitudini de sincronizare specifice unor calculatoare anterioare care aveau ceasuri separate pentru unități independente. Între memorii și unitățile pipeline sînt mai multe căi de date pentru minimizarea întîrzierilor și a conflictelor care pot interveni cînd este partajată o singură cale de date de către mai multe unități.

Plecînd din partea superioară a figurii 2.39 observăm memoriile: pentru *program* cu capacitatea de pînă la 4K cuvinte de 64 biți (timp de acces 50 ns); *scratch-pad* (S-pad) formată din 16 registre de 16 biți pentru adrese și indici; pentru *tabele* (ciclu de 167 ns, fie numai read-only, fie read/write) de pînă la 64 K cuvinte de 38 biți pentru memorarea constantelor folosite frecvent, ca sine și cosine la calcularea transformatei Fourier;

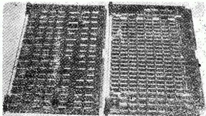


Fig. 2.38 Două plăci cu circuite ale calculatorului FPS AP-120B. Stînga : Placa cu bufere a unității de control, unde se decodifică instrucțiunile ; dreapta : o placă a memoriei program. (Fotografie furnizată de Dl. Heas și Floating Point Systems SA Ltd.)

două grupuri (*data pad X* și *data pad Y*) de 32 registre pe 38 biți pentru memorarea rezultatelor temporare în virgulă mobilă ; și pentru *date* reprezentate pe 38 biți (plus trei biți de paritate), adresabilă direct pînă la 64 Kcuvinte, iar cu o adresă suplimentară de pagină de 4 biți expandabilă la 1 Mw. La fiecare din cele două intrări ale sumatorului și multiplicatorului în virgulă mobilă există căi separate pentru date pe 38 biți. Aceste 4 căi independente pot fi alimentate de memoria principală, data pads sau memoria pentru tabele. Trei căi suplimentare pot conecta ieșirile la intrări sau la data pads sau memoria principală. Căile multiple permit citirea unui operand de la fiecare data pad și scrierea unui rezultat în fiecare data pad în cursul unui ciclu mașină. Adresele din interiorul ambelor data pads sînt furnizate de registrele de adresare a data pad (DPA). Este posibilă adresarea relativă (−4 la +3), în cadrul cîmpurilor de index a instrucțiunii XR, YR, XW, YW (vezi §2.4.4).

Memoria principală pentru date este disponibilă în module de 8K (sau module de 32K, funcție de tipul capsulei), organizate fiecare ca o pereche de blocuri de memorie independente, un bloc pentru adresele pare și unul pentru cele impare. Memoria standard are un timp de acces de 500 ns, iar memoria rapidă opțională un ciclu de 333 ns. Referințele succesive la același bloc de memorie (de exemplu, toate adresele pare mai mici ca 8K) trebuie separate prin cel puțin 3 perioade de ceas la memoria standard sau două perioade la cea rapidă. Două referințe succesive la blocuri diferite de memorie (de exemplu, două adrese vecine care se află în blocuri diferite sau două adrese de același tip separate prin 8K și, deci, aflate tot în module diferite) pot avea loc în cursul unor perioade succesive de ceas. Alternînd referințele la blocurile de memorie de adrese pare, respectiv impare, ca la accesarea secvențială a elementelor unui vector lung, se poate executa o referință într-o perioadă de ceas în cazul memoriei rapide, obținîndu-se un acces la același bloc la fiecare 333 ns și un timp de acces efectiv minim între cererile la memorie de 167 ns. Pentru memoria standard această viteză trebuie înjumătățită, obținîndu-se în cazul optim al accesului secvențial 333 ns. Dacă se accesează repetat același bloc, se înregistrează un timp de 500 ns (trei perioade de ceas). Producătorul spune că memoria are un ,,ciclu" *întrețesut* (interleaved) de 167 ns pentru memoria rapidă și de 333 ns pentru memoria standard, chiar dacă capsulele de memorie au un ciclu fizic de 333 și, respectiv, 500 ns. Cînd se fac com-

parații cu alte mașini trebuie să ne amintim că am ales timpul de acces al capsulei de memorie ca o măsură a calității memoriei (de exemplu, memoria principală la CRAY X-MP cu 38 ns, deși aceasta este organizată în blocuri pentru a produce un „ciclu" întrețesut de 9,5 ns). Dacă o referință se face la o zonă de memorie ocupată, mașina oprește execuția pînă ce se poate satisface cererea.

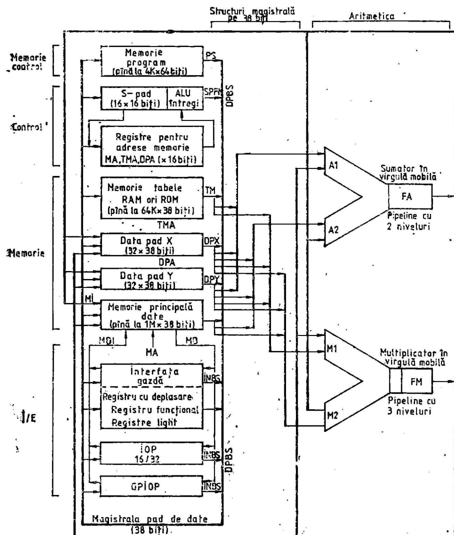


Fig. 2.39 Arhitectura calculatorului FPS AP-120B, care prezintă memoriile multiple, unitățile aritmetice pipeline și căile de date (Schema pusă la dispoziție de Floating Point Systems Inc.)

La AP-120B instrucțiunile sînt pe 64 biți, fiecare controlînd funcționarea tuturor unităților mașinii. Astfel, există, în acest sens, numai o instrucțiune în setul de instrucțiuni (vezi 2.5.4) cu cîmpuri care controlează

fiecare din cele 10 funcții, deși unele cîmpuri se suprapun, excluzînd astfel anumite combinații de funcții. Acest mod se numește „microcod orizontal". Instrucțiunile sînt executate la viteza maximă de una pe perioada de ceas, adică 6 mil. instrucțiuni pe secundă dar, deoarece fiecare instrucțiune determină execuția mai multor operații, echivalentul pe o mașină convențională unde o instrucțiune controlează numai o unitate este mai mare.

S pad conține o unitate aritmetico-logică pe 16 biți independentă folosită pentru calculul adreselor ,numărarea buclelor și a indicilor în cadrul grupului propriu de 16 biți. Se pot executa operațiile : adunare între întregi, scădere ; AND, OR ; deplasări și negări pentru utilizarea în transformata Fourier (vezi §5.5). Toate aceste operații durează o perioadă de ceas. Nu se poate executa înmulțirea. Aceste operații cu numere întregi se execută în paralel cu calculele în virgulă mobilă din unitățile pipeline.

Ambele unități pipeline de adunare și înmulțire în virgulă mobilă pot prelua o nouă pereche de operanzi și furniza un rezultat în cursul fiecărei perioade de ceas. Astfel, viteza maximă de generare a rezultatelor este de 6 Mflop/s pe pipeline, sau 12 Mflop/s dacă ambele pipeline pot fi alimentate continuu cu date. Pentru problemele obișnuite performanța se află probabil în domeniul 4—8 Mflop/s (vezi §2.5.6). Unitatea pipeline de înmulțire are lungimea de trei perioade de ceas (500 ns) iar cea de adunare de două perioade (300 ns). Unitatea pipeline de adunare poate realiza și operațiile logice AND, OR și echivalent să calculeze valoarea absolută, să convertească numerele între reprezentarea semn-valoare absolută și complementul față de 2. Fig. 2.40 și 2.41 clasifică modul de operare al unităților pipeline de adunare și înmulțire, prezentînd totodată posibilele surse și destinații ale operanzilor, cele două registre de intrare (A1, A2 sau M1, M2), operațiile parțiale executate în fiecare etaj al unităților pipeline și regis-

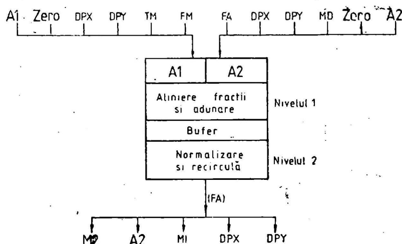


Fig. 2.40 Unitatea pipeline de adunare în două etaje, de la FPS AP-120B. Se observă sursele posibile de operanzi și destinațiile rezultatelor, cu notația din fig. 2.39. Între etaje se află bufere pentru memorarea rezultatelorin termediare. (Schema furnizată de Floating Point Systems.)

trele tampon care păstrează rezultatele parţiale intermediare. AP-120B nu posedă un divizor hardware, de aceea împărţirea în virgulă mobilă se realizează software prin aproximare polinomială.

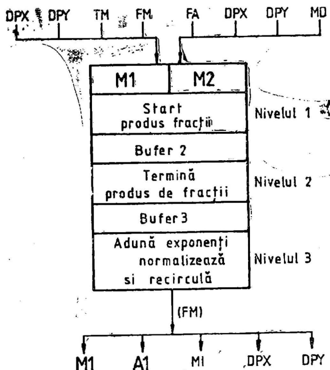


Fig. 2.41 Unitatea pipeline de înmulţire cu trei etaje, de la FPS AP-120B. Notaţia este cea folosită în fig. 2.39. (Schema furnizată de Floating Point Systems Inc.)

Operaţiile aritmetice în virgulă mobilă se execută în format pe 38 biţi (10 biţi pentru exponent şi 28 pentru mantisa în complement faţă de 2) Această reprezentare se face în domeniul $10^{\pm 153}$, cu o precizie de 8 cifre zecimale. Precizia este semnificativ mai bună decît la IBM, unde se foloseşte formatul pe 32 biţi ($10^{\pm 78}$ şi şase cifre zecimale). Se folosesc şi cifre de siguranţă pentru a minimiza pierderea de precizie în cursul execuţiei operaţiilor aritmetice. Conversia numerelor între formatul folosit de calculatorul gazdă şi formatul intern al lui AP-120B are loc odată cu transferul numerelor între cele două maşini.

Operaţiile de intrare/ieşire se execută de un port de I/E (IOP) sau de un port general programabil (GIOP). IOP accesează memoria principală a lui AP-120B prin „furturi" de cicluri şi realizează transferuri de 16 sau 32 biţi. În interiorul calculatorului se realizează transferul datelor la viteza de 1,5 Mw/s, iar în exterior la 1,3 Mw/s. Portul pe 16 biţi este folosit ca intrare analog/digitală, ieşirea datelor la display şi alte periferice standard. Portul pe 38 biţi conţine un sumator şi poate fi folosit pentru o multitudine de conversii format. Acest port poate fi folosit la conectarea între

două calculatoare AP-120B. IOP ocupă o placă logică și poate conecta pînă la 256 dispozitive externe. GIOP este un canal de I/E programabil care poate realiza un transfer continuu la viteza de 3 Mw/s cu posibilitatea conversiei formatului „în zbor", inclusiv din virgulă fixă în virgulă mobilă. Conține două microprocesoare de 18 Mips și ocupă trei plăci logice. În mod obișnuit este folosit pentru interfațarea cu discuri, dispozitive de afișare în timp real, camere video și alte calculatoare.

Arhitectura calculatorului AP-120B cu memoria de date standard de 64 K poate fi scrisă în notație ASN (vezi §1.2.4):

$$C(AP-120B)=I1[(Fp38(+),\ Fp38(*))_{\overline{7\times 38}}$$
$$(P1,\ M1\text{-}a,\ M2,\ M3,\ P2)]_h$$

$$I1=\{I_{64}^{167}-M4\}$$

$$P1(S\text{-}pad)=B_{16}-M_{16*16}$$

$$M1(\text{memoria de date})=8\{2M_{4K*38}^{500}(MOS)\}$$

$$M2(\text{tabele})=M_{64K*38}$$

$$M3(X,\ Y\ \text{data pads})=2M_{32*38}$$

$$M4(\text{program})=M_{4K*64}^{50}(\text{bipolar})$$

$$P2(IOP)=256D-IO_{16}-a$$

2.5.3. Tehnologia

AP-120B este proiectat să fie fiabil, de aceea s-au folosit numai componente și tehnologii bine verificate, la condiții limită de funcționare foarte clare. Ca rezultat, timpul mediu între două defecțiuni (mean time between failure -MTBF) este în mod obișnuit de la cîteva luni, la un an. Calculatorul este construit cu circuite TTL Schottky, cu un nivel de integrare ce variază de la cîteva porți pe capsulă la cîteva sute. Timpul de propagare normal pentru o poartă este de 3—5 ns. Diverse registre interne folosesc aceeași tehnologie. Acestea sînt registrele S-pad, data-pad, ca și stiva cu adresele de revenire din subrutine. Memoria program cu timp de acces de 50 ns, ca și memoria pentru tabele cu timp de acces de 167 ns folosesc circuite bipolare Schottky de 1 K, în timp ce memoria pentru date, mai mare și mai lentă, folosește circuite MOS de 4 K sau 16 K.

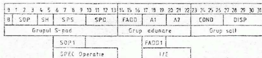
Este interesant de comparat CRAY X-MP și AP-120B din punctul de vedere al tehnologiei, vitezei și consumului de putere, deoarece cele două calculatoare reprezintă două extreme opuse. CRAY X-MP folosește tehnologie ECL de mare viteză, dar și cu un consum mare de putere, perioada ceasului fiind de 9,5 ns. De aici, necesitatea unui sistem de răcire cu freon, pentru disiparea unui total de 115 KW. Pe de altă parte, AP-120B folosește tehnologie cu un consum redus de putere și, în consecință, are perioada ceasului de 167 ns. Totuși, puterea totală consumată de 1,3 KW permite folosirea răcirii cu aer.

2.5.4. Setul de instrucţiuni

AP-120B nu are instrucţiuni vectoriale *per se*. În fiecare perioadă de ceas se execută o instrucţiune de 64 biţi, care are cîmpuri de control pentru toate unităţile. Ca exemplu, dacă cîmpul FM (vezi mai jos) care controlează unitatea pipeline de înmulţire este activat (bitul 51 al instrucţiunii este 1), atunci toate datele aflate în acest pipeline sînt deplasate la următorul etaj. Prin urmare, unitatea pipeline trebuie activată trei perioade de ceas pentru a „împinge" o pereche de argumente prin cele trei etaje şi a obţine, astfel, un rezultat. Pentru fiecare element al unui vector, care ar urma, unitatea trebuie activată o perioadă de ceas. Cel mai probabil, acest proces se va realiza cu o buclă. În fig. 2.42 se prezintă formatul general al instrucţiunii. Notaţiile pentru sufsele şi destinaţiile datelor corespund fig. 2.39. Descrierea completă a instrucţiunii se găseşte în AP-120B Processor Handbook (FPS 1976a). Pentru a indica modul de operare al unei instrucţiuni, dăm în continuare cîteva exemple de utilizare a cîmpurilor de date:

(1) *grupul S-pad* (controlează unitatea ALU ce efectuează calcule cu întregi pe 16 biţi şi registrele aferente)

SOP specifică o operaţie S-pad diadică, de exemplu, ADD, SUB, MOV, AND, OR, EQUIV; operanzii sînt registrele SPS şi SPD, iar rezultatul trece în SPD;



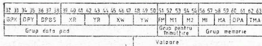


Fig. 2.42 Cîmpurile de date ale unei instrucţiuni pe 64 biţi de la FPS AP-120B. Această instrucţiune controlează funcţionarea tuturor unităţilor calculatorului la fiecare perioadă de ceas. (Diagrama furnizată de Floating Point Systems Inc.)

SPS numărul (pe 4 biţi) al registrului sursă pe 16 biţi;

B dacă este 1, biţii registrului sursă sînt inversaţi înaintea execuţiei operaţiei;

SPD numărul registrului destinaţiei pe 16 biţi;

SH specifică deplasarea simplă la stînga sau dreapta, sau deplasare dublă la dreapta după calcul;

SOP1 specifică operaţii monadice cu datele din registrul SPD, cînd SOP=0, de exemplu incrementare (+1), decrementare (−1) sau complementarea registrului SPD;

SPEC instrucţiuni de control, condiţionale sau salt.

(2) *grupul pentru adunare în virgulă mobilă*

FADD specifică operaţii diadice, de exemplu FADD, FSUB, AND, OR, EQUIV cu date în A1 şi A2. Rezultatele intermediare se deplasează un segment prin pipeline pînă la următorul registru tampon;

A1 sursa operandului ce se încarcă în primul registru de intrare în sumator, de exemplu FM (ieşirea multiplicatorului), DPX, TM;

A2 sursa operandului ce se încarcă în al doilea registru de intrare;

FADD1 specifică operaţii monadice asupra datelor din A2 cînd FAD= =0, de exemplu converteşte A2 în întreg, format semn-valoare absolută, sau complement faţă de 2; ia valoare absolută.

(3) *grupul de I/E* (controlează operaţiile de I/E şi transfer cu magistralele), de exemplu:

DPBS→SPD conţinutul magistralei data-pad trece în S-pad

DPBS→TMA conţinutul magistralei data-pad se încarcă în registrul de adrese al memoriei cu tabele

SPFN — PNLBS ieşirea S-pad conectată la magistrala panel;

INTA acceptarea întreruperi; adresa dispozitivului este plasată în DPBS.

(4) *grupul pentru efectuarea salturilor*

COND condiţia pentru salt, de exemplu întotdeauna, pe condiţie, pe eroare aritmetică, revenire din subrutină, deci cînd FA sau SPFN=, ≠, ⩾, > 0, 0,;

DISP dacă se efectuează saltul, adresa următoare este adresa curentă +DISP−16, salt relativ de la −16 la +15.

(5) *grupul data-pad* (controlează transferurile cu data pads X şi Y)

DPA adresa data-pad curentă;

DPX încarcă data-pad X din DPBS, FA sau FM;

DPY încarcă data-pad Y din DPBS, FA sau FM;

DPBS selectează DPX, DPY, MD, SPFN sau TM pentru a depune conţinutul pe magistrala data-pad;

XR conţinutul registrului data-pad cu adresa DPA+XR-4 este trimis în DPX;

YR conţinutul registrului data-pad cu adresa DPA+YR-4 este trimis în DPY;

XW DPX este transferat în registrul data-pad cu adresa DPA+ XW-4;

YW DPY este trimis în registrul data-pad cu adresa DPA+YW-4.

(6) *grupul pentru înmulţire în virgulă mobilă*

FM înmulţire sau nici o operaţie. Rezultatele intermediare se deplasează un etaj prin pipeline pînă la următorul registru tampon;

M1 se încarcă din FM, DPX, DPY sau TM;

M2 se încarcă din FA, DPX, DPY sau MD.

(7) *grupul pentru memorie* (controlează transferurile cu memoriile de date şi cu tabele)

MI încarcă registrul de intrare în memorie din FA, FM sau DPBS;

MA   incrementează sau descrementează registrul de adrese cu 1, sau citeşte din SPFN şi iniţiază ciclu la memoria de date;

DPA   incrementează sau decrementează adresa de data-pad cu 1 sau ia adresa din SPFN;

TMA   ca şi DPA dar pentru memoria cu tabele.

2.5.5. Software

Soft-ul pentru AP-120B, exceptînd driver-ele, este scris în FORTRAN, astfel că poate fi compilat pe diverse calculatoare gazdă. Poate fi împărţit în următoarele categorii:

(1) sistem de operare;
(2) soft de dezvoltare programe;
(3) biblioteci de aplicaţii.

Sistemul de operare constă dintr-un executiv APEX şi un set de rutine de diagnoză APTEST. Executivul controlează transferul datelor între gazdă şi AP-120B, transferă programele AP de la gazdă în memoria de programe sursă AP (PS) şi iniţiază execuţia programelor AP. Fig. 2.43 ilustrează modul de operare al APEX. Cele mai multe programe utilizator sînt în FORTRAN şi pot apela fie biblioteca matematică AP-120B, fie subrutinele scrise de utilizator cu instrucţiuni AP-120B, pentru a manipula masivele. Şi APEX este o subrutină inclusă în programul FORTRAN compilat şi executat pe calculatorul gazdă. Conţine un tabel cu adresele şi conţinutul rutinelor AP deja încărcate în memoria de program sursă AP. Instrucţiunile AP-120B ale fiecărei subrutine sînt memorate în memoria calculatorului gazdă. Presupunînd că s-au apelat deja subrutinele 1 şi 2, la execuţia programului FORTRAN au loc următoarele evenimente:

(1) programul FORTRAN apelează AP cu rutina (VADD);
(2) rutina 3 apelează APEX;
(3) se caută în tabelul memoriei PS; rutina 3 nu este găsită;
(4) APEX transferă instrucţiunile AP-120B de la gazdă la memoria PS;
(5) se actualizează tabelul din memoria PS;
(6) APEX iniţiază execuţia rutinei 3 în AP-120B.

Datele se transferă de la calculatorul gazdă la AP-120B prin apelul subrutinei APPUT, iar rezultatele se obţin cu subrutina APGET. Ambele subrutine apelează APEX pentru controlul transferului. Odată începută execuţia unui program pe AP, APEX redă controlul programului FORTRAN apelant, care poate continua pe calculatorul gazdă execuţia altor calcule, care nu folosesc AP. Dacă se întîlneşte un apel de subrutină AP, înaintea terminării celei anterioare, APEX va aştepta terminarea acesteia.

Soft-ul de dezvoltare constă din:

(1) *Biblioteca matematică* — peste 250 subrutine FORTRAN pentru operaţii cu masive. Subrutinele, scrise în limbaj de asamblare, sînt atent optimizate. Iată cîteva exemple:

VADD, VMUL   adunare, respectiv înmulţire element-cu-element;
SVE, DOTPR   suma, respectiv produsul elementelor vectorului;
MMUL, MATINV   înmulţirea şi inversarea matricelor;
CFFT, ACORT transformata Fourier rapidă complexă şi auto corelaţia.

În plus, există biblioteca matematică avansată care conține rutine pentru generarea de funcții, căutare binară, tridiagonalitate, diagonalizare rezolvarea sistemelor de ecuații reale sau complexe rare, rezolvarea ecuațiilor diferențiale ordinare prin integrare Runge-Kutta.

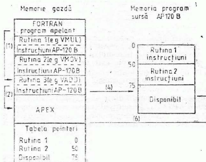


Fig. 2.43 Diagrama ce prezintă modul de acțiune al executivului APEX, în cursul execuției unui program (furnizată de Floating Point Systems Inc.). Atît instrucțiunile cît și datele sînt transferate între calculatorul gazdă și AP-120B, sub controlul lui APEX.

(2) APAL asamblorul care asamblează programe pe calculatorul gazdă pentru a fi executate pe AP.
(3) APLOAD — Leagă modulele obiect APAL separate într-un singur modul ce poate fi executat pe AP.
(4) APSIM, APDBUG — simulează un program AP și permite depanarea pe gazdă sau, respectiv, AP.
(5) VFC — înlănțuie funcții vectoriale. Transformă apelurile multiple la biblioteca matematică într-un singur apel, reducînd astfel overhead-ul.
(6) FORTRAN AP — un compilator ce se execută pe calculatorul gazdă, acceptînd FORTRAN IV, și produce cod direct executabil pe AP.

Soft-ul amintit permite pregătirea programelor într-o multitudine de moduri. Pentru scăderea vitezei de execuție și creșterea ușurinței de pregătire, se folosesc: asamblorul APAL, VFC, rutinele matematice și FORTRAN AP.

Bibliotecile de aplicații disponibile sînt:
(1) SIGLIB — biblioteca pentru prelucrarea semnalelor cu rutine pentru histograme, corelații, funcții de transfer și coerență etc.
(2) IMP — biblioteca pentru prelucrarea imaginilor cu rutine pentru transformata Fourier rapidă 2D, convoluții, filtrare etc.
( ) AMLIB — biblioteca matematică avansată cu rutine pentru generarea de funcții, integrare Runge-Kutta, rezolvarea matricelor rare și calculul vectorilor proprii.

2.5.6. Performanța

AP-120B nu posedă ceas de timp real, de aceea este imposibil să se evalueze cu acuratețe timpii de execuție ai programelor. Folosirea ceasului calculatorului gazdă nu oferă o măsură precisă datorită sistemului de operare al calculatorului gazdă. De aceea, pentru evaluarea performanței sîntem obligați să folosim formulele furnizate în biblioteca matematică. Datele (FPS 1976 b) pe care le folosim corespund unor formule de calcul al timpului care pot fi legate direct de formula (1.4a), care definește r_∞ și $n_{1/2}$. Diferențele minore ce apar în materiale ulterioare (FPS 1979a, b) sînt lipsite de importanță. Datorită naturii sincrone a acestor mașini, formulele teoretice ar trebui să fie valabile; totuși, absența unui ceas face optimizarea programelor mari foarte dificilă. Evaluarea timpilor de execuție devine în acest caz plicticoasă și supusă erorilor. O alternativă este de a simula execuția unui program AP pe calculatorul gazdă folosind programul APSIM (vezi §2.5.5). Acest program furnizează timpul de execuție teoretic dar, din nou, nu poate fi folosit la evaluarea programelor mari, deoarece se execută de aproximativ 1000 ori mai lent decît programul propriu-zis pe AP-120B.

Folosind documentația bibliotecii matematice (FPS 1976b) vom prezenta formulele pentru o serie de operații vectoriale simple, de unde vom estima valori probabile pentru $n_{1/2}$ și r_∞. Stabilim formula pentru memoria standard (timp de acces 500 ns) și dăm valorile îmbunătățite ale lui r_∞ pentru memoria rapidă (timp de acces 333 ns) în paranteze. Unde există o variație mică datorită alegerii pentru vectori a unor locații pare sau impare, am considerat timpul minim. Nici una din aceste alternative nu modifică substanțial caracterul mașinii și, de aceea, pot fi ignorate.

(1) *deplasare vector* **CALL VMOV (A, I, C, K, N)**

$$C_{mK} \leftarrow A_{mI}, \quad m = 0, 1, \ldots N\text{-}1$$

Deplasarea vectorului A în C, în memorie. K și I sînt incremenții între elementele succesive ale lui A, respectiv C. Pentru N operații timpul este:

$$t = \frac{2}{3}(N + 1)\mu s,$$

de aici,

$$r_\infty = 1{,}5\,(3{,}0) \text{ Mop/s}, \quad n_{1/2} = 1$$

Observăm că această operație este cu restricție de memorie și că viteza de transfer se dublează la memoria rapidă. Oricum, $n_{1/2}$ nu este afectat de tipul memoriei.

(2) *adunare vectorială* **CALL VADD (A, I, B, J, C, K, N)**

$$C_{mK} \leftarrow B_{mJ} + A_{mI}, \quad m = 0, 1, \ldots N-1$$

Timpul este pentru N operații

$$t = N + 1\ \mu s$$

prin urmare,

$$r_\infty = 1(2) \text{ Mflop/s}, \quad n_{1/2} = 1$$

(3) *înmulțire vectorială* CALL VMUL (A, I, B, J, C, K, N)

$$C_{mK} \leftarrow B_{mJ} * A_{mI}, \quad m = 0,1,\ldots,N-1$$

Timpul de execuție pentru N operații este

$$t = N + 2 \ \mu s$$

deci,

$$r_\infty = 1(2) \ \text{Mflop/s}, \quad n_{1/2} = 2$$

(4) *împărțire vectorială* **CALL VDIV (A, I, B, J, C, K, N)**

$$C_{mK} = B_{mJ}/A_{mI}, \quad m = 0,1,\ldots,N-1$$

Pentru N operații

$$t = 1{,}83(N + 3)\mu s$$

deci,

$$r_\infty = 0{,}55(0{,}55) \ \text{Mflop/s}, \quad n_{1/2} = 3$$

și observăm că, deoarece operația este dominată de calcule aritmetice, memoria mai rapidă nu determină creșterea performanței.

(5) *exponent vectorial* **CALL VEXP (A, I, C, K, N)**

$$C_{mK} \leftarrow \exp(A_{mI}), \quad m = 0,1,\ldots N-1$$

Pentru N exponențiale,

$$t = 4{,}87(N + 0{,}3) \ \mu s$$

deci,

$$r_\infty = 0{,}2 \ \text{Mflop/s}, \quad n_{1/2} = 0{,}3$$

(6) *produsul* **CALL DOTPR (A, I, B, J, C, N)**

$$C \leftarrow \sum_{m=0}^{N-1} A_{mI} * B_{mJ}$$

Pentru efectuarea a 2N operații

$$t = \frac{2}{3}(N + 2) \ \mu s$$

deci,

$$r_\infty = 3(6) \ \text{Mflop/s}, \quad n_{1/2} = 2$$

Timpii de execuție și performanțele selectate, care pot fi considerate tipice pentru operațiile vectoriale de tipul memorie-la-memorie, arată că în cazul acestor operații se poate realiza numai o fracție mică din performanța potențială de 12 Mflop/s. Cauza este lărgimea de bandă a accesului la memorie insuficientă pentru a susține o viteză de calcul de valoare mare. O operație vectorială simplă are, deci, operanzi vectoriali și produce un vector rezultat. Prin urmare, o viteză de calcul de 12 Mflop/s necesită o lărgime de bandă de 36 Mw/s. Lărgimea de bandă a memoriei standard variază între 2 și 3 Mw/s, funcție de condiția dacă toate referințele sînt la același bloc sau alternează între blocuri diferite de memorie. Memoria ra-

pidă are lărgimea de bandă între 3 şi 6 Mw/s. Astfel, este evident că lărgimea de bandă este aproximativ 1/10 din cea necesară atît la adunare, cît şi înmulţire, cînd datele se află în memoria principală.

Deci, restricţia care împiedică atingerea unor viteze de prelucrare mari de către AP-120B este lărgimea de bandă mică a accesului la memorie. Pentru a evita ca aceasta să devină o strangulare, este necesar ca intensitatea calculelor (vezi p. 106), f, să fie de cel puţin 2 operaţii în virgulă mobilă pe referinţă la memorie (flop/ref) în cazul memoriei rapide, sau cel puţin 4 flop/ref în cazul memoriei standard. Aceste condiţii sînt îndeplinite pentru algoritmi mult mai complecşi şi dimensiuni, n, suficient de mari ale problemelor : de exemplu, înmulţirea matricelor, $f = 2n/3$ flop/ref ; transformata Fourier rapidă (FFT), $f = 1{,}25 \log n$ flop/ref.

Este bine să examinăm mai în detaliu algoritmul FFT, deoarece este algoritmul principal pe care îl execută maşini ca AP-120B. Pe tot parcursul algoritmului, operaţiile sînt de tipul „fluture" (vezi ecuaţia (5.87)) :

$$c = a + wb \tag{2.17a}$$

$$d = a - wb \tag{2.17b}$$

unde a şi b sînt elemente complexe aflate în memorie, iar c şi d sînt rezultate complexe ce urmează să fie scrise în memorie. Constanta complexă w poate să se afle într-un registru .Ecuaţia (2.17) necesită o înmulţire complexă ($s = w*b$) şi două adunări complexe ($a + s$, $a - s$) pentru 4 referinţe la memorie. Este echivalentul a 10 operaţii aritmetice reale pentru 8 referinţe la memorie pentru numere reale, deci $f = 1{,}25$ flop/ref. Consideraţiile precedente sînt valabile pentru baza 2 şi arată că acest algoritm nu are o intensitate a calculelor suficient de mare pentru a menţine unităţile aritmetice pipeline ocupate. Prin combinarea a 2 nivele FFT, obţinem algoritmul în baza 4 şi o creştere a intensităţii calculelor la 2,5 flop/ref, care satisface condiţiile enunţate pentru memoria rapidă. Subrutina CFFT din biblioteca matematică foloseşte acest ultim algoritm pentru care se atinge performanţa de 8 Mflop/s.

Valorile lungimii performanţei-jumătate se află în domeniul 1—3, ceea ce arată că AP-120B, deşi are multe caracteristici de paralelism, se comportă foarte asemănător cu un calculator serial. Din acest punct de vedere, calculatorul este similar lui CRAY-1 ($n_{1/2} \approx 10$) şi destul de diferit de CYBER 205 ($n_{1/2} \approx 100$), sau ICL DAP ($n_{1/2} \approx 1000$). Alegerea celui mai bun algoritm se face pe baza lui $n_{1/2}$ (vezi capitolul 5) şi ne putem aştepta ca algoritmi optimizaţi pentru un calculator serial să furnizeze performanţe bune şi pe AP-120B. Oricum, aşa cum s-a accentuat, performanţa unui program poate depinde mai mult de gestiunea referinţelor la memorie, decît de lungimea vectorului de care ţine cont $n_{1/2}$.

2.5.7. FPS-164 (redenumit M140 şi M30) şi 264 (redenumit M60)

Aşa cum se poate vedea în fig. 2.44, FPS-164 este substanţial mai mare decît AP-120B, avînd o înălţime de aproximativ 1,65 m şi ocupind o suprafaţă de $0{,}75 \times 2{,}1$ m^2, în special datorită memoriei mai mari. Pentru FPS-164/MAX şi FPS-264 se foloseşte acelaşi cabinet. Principalele îmbunătăţiri introduse la FPS-264 (în comparaţie cu AP-120B) sînt;

(a) aritmetica în virgulă mobilă pe 64 biţi, în comparaţie cu 38;
(b) aritmetica numerelor întregi pe 32 biţi, în comparaţie cu 16;
(c) adrese pe 24 biţi pentru maximum 16 Mw în comparaţie cu adresarea a numai 64 Kw cu 16 biţi;
(d) registrele X- şi Y-pad pe 64 biţi în comparaţie cu 32;
(e) 64 registre de adrese S-pad pe 32 biţi în comparaţie cu 16 pe 16 biţi;
(f) cache pentru 1024 instrucţiuni pe 64 biţi care înlocuieşte memoria program;
(g) stiva formată din 256 registre pe 32 biţi pentru adresele de revenire din subrutine;
(h) memoria centrală expandabilă de la 0,25 la 7,25 Mw, cu protecţie;
(i) memorie pentru tabele de 32 Kw RAM;
(j) un ceas pentru evaluarea timpilor de execuţie a programelor — care din păcate nu există la AP-120B.

Creşterea preciziei operaţiilor aritmetice şi a domeniului de adresare deplasează clasificarea acestui calculator din domeniul minicalculatoarelor în cel al calculatoarelor mari curente. Modul cum se regăsesc aceste aspecte în arhitectură se poate vedea în fig. 2.45. Spre deosebire de AP-120B,

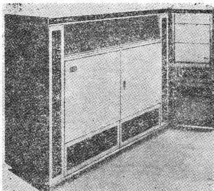


Fig. 2.44 Calculatorul FPS-164

atît instrucţiunile cît şi datele se află în memoria centrală, cînd este necesar instrucţiunile fiind încărcate în memoria cache. Prin urmare, aceasta înlocuieşte memoria separată pentru programe de la AP-120B. La FPS-164 apelul subrutinelor se face mai eficient prin includerea stivei pentru subrutine, care poate memora pînă la 256 adrese de revenire pe 32 biţi.

Memoria pentru tabele, denumită şi memoria auxiliară, a devenit o memorie R/W, folosită ca registre temporare pentru rezultatele intermediare. Totuşi, primii 8 K ai acestei memorii sînt rezervaţi pentru constante care numai se citesc, din care 5 K sînt pre-înscrişi, iar următorii 16 K sau 32 K (funcţie de opţiune) pot fi folosiţi de programele utilizator. Memoria principală este organizată în module, fiecare cu blocuri de memorie pare

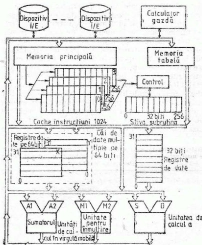


Fig. 2.45 Arhitectura calculatorului FPS-164.

şi impare, ca la AP-120B. Varianta FPS-164 iniţială a folosit circuite NMOS dinamice de 16 Kb, cele 12 module de memorie cuprind 24 plăci logice (fiecare placă un bloc), obţinîndu-se un total de 1,5 Mw. Utilizarea ulterioară a circuitelor de memorie de 64 Kb a permis creşterea capacităţii memoriei la 7,25 Mw. Memoria principală lucrează ca un pipeline cu trei etaje, cererile succesive la acelaşi bloc avînd posibilitatea de a interveni la fiecare perioadă de ceas. Deoarece memoria pentru tabele lucrează ca un pipeline cu două etaje, cererile succesive pot interveni în cursul ciclurilor de ceas succesive. O nouă caracteristică a memoriei este maparea şi protecţia. În aceste condiţii, conţinuturile registrului MDBASE se adună la adresa depusă pe linia MA (vezi fig. 2.39), formînd o adresă fizică. Dacă aceasta depăşeşte conţinuturile registrului MDLIMIT, nu va avea loc nici un acces la memorie: o citire produce zerouri, iar scrierea este ignorată.

FPS-16P are perioada nominală a ceasului de 167 ns, ca AP-120B. De fapt, perioada reală a ceasului este ceva mai mare, de 182 ns, ceea ce conduce la o viteză asimptotică de 11 Mflop/s. Mașina mai are un ceas de timp real programabil și un timer CPU care se incrementează la fiecare perioadă de ceas. Acestea permit o evaluare corectă a timpilor de execuție pentru programele utilizator.

Performanța aritmetică teoretică este de 11/12 din valoarea teoretică furnizată pentru AP-120B. Timer-ul CPU permite evaluarea programelor de test cu acuratețe, noi citînd rezultatele obținute de Thompson (comunicare personală). În tab. 2.8 se prezintă rezultatul testului (r_∞, $n_{1/2}$) pentru operații aritmetice memorie-la-memorie. Se compară rezultatele codului FORTRAN pur (valoarea superioară a perechii) cu cele ale rutinelor optimizate din biblioteca matematică. Așa cum ne așteptăm din evaluările teoretice de la AP-120B, operațiile diadice se execută cu aproximativ 1 Mflop/s. Exceptînd operația de adunare, nu este nimic care să recomande utilizarea rutinelor din bibliotecă, deoarece aceeași performanță se obține cu FORTRAN, eliminîndu-se totodată overhead-ul, cum se poate vedea din valorile mai mici ale lui $n_{1/2}$. Execuția unei operații diadice este situația cea mai defavorabilă pentru performanță, deoarece nu există nici o posibilitate de a memora rezultate intermediare în registrele cu acces rapid, iar numărul operațiilor în virgulă mobilă pe referință la memorie, f, este de numai 1/3. Ultimele: două cazuri al triadei și al celor 3 operații

Tabelul 2.8. **Rezultatele testului (r_∞, $n_{1/2}$) pentru un FPS-164 care execută operații cu date aflate în memorie. Literele mari notează vectori, literele mici scalari (Date puse la dispoziție de Bill Thompson- TUCC).**

Operația : instrucțiunea 10 program (1.5)	r_∞ (Mflop/s)	$n_{1/2}$
A = B + C	0,88	5
CALL VADD	1,06	16
A = B × C	1,07	5
CALL VMUL	1,04	17
A = B/C	0,30	7
A = bx (C − D)		
OPTC = 1	0,8	—
OPTC = 3	3,4	—
A = B + Cx (D E)		
OPTC = 1	1,0	
OPTC = 3	3,2	—

ridică valoarea lui f la 2/3, respectiv 3/4. Cu nivelul de optimizare al compilatorului (OPTC) egal cu unu, acesta executînd numai optimizare locală, se obține o performanță diadică de aproximativ 1 Mflop/s. Dacă OPTC=3, cînd se folosește conceptul software de pipelining prin care se suprapun operațiile executate, performanța crește de trei ori.

În tab. 2.9 se prezintă performanțele unor teste mai importante, buclele Livermore și **LINPACK**. În general, performanța asociată unei probleme obișnuite se situează între 1 și 5 Mflop/s, funcție de atenția acordată programării problemei.

FPS-264 este o versiune în tehnologie ECL a arhitecturii FPS-164, care permite reducerea perioadei ceasului de la 182 ns la 53 ns, un raport de 3,4. Alte îmbunătăţiri ale memoriei cache şi a memoriei centrale au condus la o creştere a performanţei de 4 pînă la 5 ori faţă de FPS-164. Se folosesc circuite ECL de 100 K produse de Fairchild (faţă de Schottky TTL la FPS-164), iar memoria centrală foloseşte capsule NMOS static de 64 Kb (în comparaţie cu NMOS dinamic la FPS-164). O placă de memorie conţine 0,5 Mw, împărţite în două blocuri. La prima maşină memoria centrală maximă era de 4,5 Mw împărţite în 18 blocuri. Memoria cache de 8 Kw este formată din două blocuri întreţesute de 4 Kw, în comparaţie cu cea de 1 Kw de la FPS-164.

2.5.8. FPS-164/MAX (redenumit M145)

În anul 1984 s-a anunţat o nouă versiune a calculatorului FPS-164, FPS-164/MAX, care înseamnă accelerator pentru algebra matricială (Charlesworth şi Gustafson 1986). Această maşină are un FPS-164 ca master, la care se pot adăuga pînă la 15 plăci MAX, fiecare echivalentă cu 2 CPU 164, fiecare unitate centrală avînd suplimentar 4 registre vectoriale a 2048 elemente. În total, se obţine echivalentul a 31 CPU 164 sau 341 Mflop/s.

Arhitectura plăcii MAX este prezentată în fig. 2.46. Placa are două CPU, fiecare cu un multiplicator pipeline pe 64 biţi, cu 8 etaje, care trimite rezultatele produse într-un sumator în virgulă mobilă cu 8 etaje. O intrare a fiecărui multiplicator (şi există trei multiplicatoare pe cele 15 plăci) este alimentată de memoria centrală, în timp ce cealaltă primeşte datele de la registre scalare sau vectoriale locale plăcii. În mod similar, a doua intrare în sumator este alimentată de registre locale. În modul emisie (broadcast), acelaşi număr este trimis simultan din memoria centrală tuturor multiplicatorilor. Perioada ceasului este aceeaşi ca la FPS-164, adică 182 ns, de unde performanţa maximă a fiecărei plăci de 22 Mflop/s. FPS-164/MAX foloseşte tehnologie VLSI CMOS, în timp ce unităţile aritmetice pipeline (prezentate în fig. 2.47) folosesc circuitele WEITEK.

Plăcile MAX ocupă poziţii de plăci de memorie şi este posibil să se facă dintr-un FPS-164 un FPS-164/MAX. Plăcile MAX sînt văzute de calculatorul gazdă ca ultimul Mw de spaţiu de adresare din cei 16 Mw, lăsînd o capacitate de memorie normală maxim adresabilă de 15 Mw. FPS-164/MAX foloseşte aceeaşi placă de memorie ca şi FPS-264, cu 0,5 Mw. Un FPS-164/MAX complet are 29 poziţii pentru plăci de memorie, din care 14 sînt folosite pentru cele 7 Mw de memorie fizică, iar 15 pentru cele 15 plăci MAX. Disponibilitatea circuitelor NMOS static de 256 Kb va permite realizarea unei capacităţi de 1 Mw pe placă, ceea ce ar conduce la creşterea capacităţii centrale la 15 Mw, maximum adresabil.

Ideea plăcii MAX este de a creşte viteza operaţiilor aritmetice prin încuibărirea a 2 sau 3 bucle DO, situaţie frecventă la operaţiile cu matrici, în particular la înmulţirea matricelor. În acest exemplu, cele 31 CPU 164 ale unui sistem complet ar fi folosite simultan pentru calculul a 31 produse interioare necesare pentru producerea a 31 elemente ale unei coloane a matricii produs. Arhitectura FPS este deja optimizată pentru

Tabelul 2.9. **Performanțele calculatoarelor FPS-164, 264 și FPS-164/MAX pentru cîteva teste Livermore și LINPACK. Valorile reprezintă Mflop/s pentru numere pe 64 biți.**

Problema	FPS-164	FPS-264	FPS-164/MAX
Performanța teoretică	—	—	33(1)
maximă	—	—	99(4)
	11	38	341(15)
$(r_\infty, n_{1/2})$	(1,07 ; 5)	—	—
FORTRAN § 1.3.3			
Livermore 3	3,0[e]	—	—
produs interior			
Livermore 6	1,1[e]	—	—
tridiagonal			
Livermore 14	1,5[e]	—	—
analiza particulelor			
LINPACK[a]	1,4	4,7	—
FORTRAN[b]			
n = 100			
Asamblor[c]	—	—	(6,1)[g]
bucla interioară			
n = 100	2,9	10	20(15)[g]
Matrice-vector[d]	—	—	15(1)[h]
cel mai bun asamblor			26(4)[h]
n = 300	8,7	33	—

Note

a Rezolvarea ecuației liniare cu DGEFA și DGESL pentru matrice de ordinul 100 (Dongarra et al 1979).

b Toate cod FORTRAN (Dongarra 1985).

c Rutine BLAS optimizate în asamblor (Dongarra 1985).

d Metoda matrice-vector în FORTRAN, de Dongarra și Eisenstat(1984).Matrice de ordinul 300. Asamblorul cel mai bun (Dongarra 1985).

e Gustafson 1985.

f Numărul de plăci IMAX este trecut în paranteze.

g FPS 1985 b.

h Dongarra 1986.

calculul eficient al produselor interioare, iar singura problemă este asigurarea că datele necesare sînt disponibile unităților pipeline. Un registru vectorial din fiecare CPU primește una din cele 31 linii ale primei matrici, în timp ce elementele coloanei celei de-a doua matrici sînt emise (modul broadcast), unul cîte unul, tuturor unităților centrale unde se acumulează cele 31 produse interioare, ca în fig. 2.48. Se produc 31 elemente în coloana respectivă a matricei rezultat. Se pot calcula toate celelalte coloane pentru aceleași 31 linii, fără modificarea conținutului registrelor vectoriale, numai în acest mod de minimizare a transferurilor de date fiind posibilă atingerea performanței maxime de către FPS-164/MAX. Din fericire, multe probleme de algebră liniară (rezolvarea de ecuații, calculul va-valorilor proprii etc.) pot fi formulate pentru satisfacerea acestei condiții. La acest nivel al calculului matricei produs, s-au calculat 31 linii. Acum, registrele vectoriale trebuie încărcate pentru calculul următoarelor 31 linii, pînă ce calculul matricei produs se încheie.

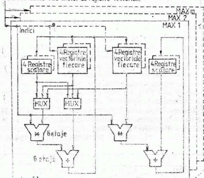


Fig. 2.46 Arhitectura plăcii acceleratorului matricial (MAX).

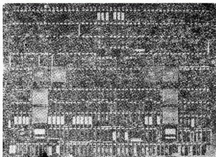


Fig. 2.47 O placă MAX.

. Nucleul algoritmului anterior este produsul matricii **A** (31×2048) cu matricea **B** (2048×2048) pentru a produce matricea **C** (31×2048). Presupunem că inițial **C** este ștearsă și transferată în memoria centrală după execuția următorului cod:

DO 1 J = 1,2048

DO 1 K = 1,2048 (2.18)

DO 1 I = 1,31

1 C(I, J) = C(I, J) + A(I, K)*B(K, J)

unde I este numărul unităţii centrale.

Bucla DO-I este implementată prin transmiterea cantităţii scalare B(K, J) tuturor celor 31 CPU, înmulţirea ei cu A(I, K) care este un element luat din vectorul local A_I, şi acumularea produsului interior în C(I, J) care este preluat de vectorul local C_I. În aceste operaţii toate cele 31 CPU lucrează sincron (lock-step), dar cu datele lor individuale (I este numărul de ordine al CPU), cu alte cuvinte controlul este de tip SIMD. Bucla DO-K acumulează produsul interior, iar bucla DO-J deplasează coloană cu coloană. Toate cele trei bucle din (2.18) pot fi executate fără a se transfera date între memoria centrală şi registrele de pe plăcile MAX, condiţia esenţială a utilizării lor eficiente. Se poate spune că pentru atingerea vitezei maxime, datele trebuie refolosite atît în spaţiu cît şi timp.

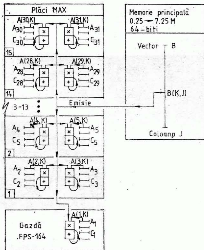


Fig. 2.48 Calculul simultan a 31 de produse interioare. ($A_I * B_J$ I = 1,31) folosind 31 plăci MAX. Elementele lui B sînt transmise unul cîte unul tuturor plăcilor, fiecare placă acumulînd două produse interioare ($S_I = S_I + A_{I,k} * B_{k,J}$; k = 1, 2048).

Reutilizarea în spaţiu se referă la transmiterea unei singure cantităţi B(K, J) simultan celor 31 CPU, în bucla DO-I, iar reutilizarea în timp se referă la faptul că C(I, J) şi A(I, K) sînt continuu reutilizate din memoria locală în buclele DO-K şi DO-J.

Scopul facilităţii de reutilizare este de a limita necesarul de transferuri între memoria centrală, şi plăcile MAX şi de a executa numărul maxim de operaţii aritmetice între astfel de transferuri, pentru a micşora penalitatea încărcării registrelor. Am exprimat anterior acest fenomen prin intensitatea calculelor, f, care este numărul de operaţii în virgulă mobilă pe referinţă la memorie. În exemplul anterior al înmulţirii matricelor, expresia 1 conţine două operaţii aritmetice, iar referinţele sînt citirea lui **A** şi **B** şi memorarea lui **C**. Astfel,

$$f = (2 \times 31 \times 2048 \times 2048)/(2110 \times 2048) = 60 \quad (2.19)$$

Această valoare poate fi comparată cu parametrul hardware ($f_{1/2}$ care este jumătate din raportul performanţei aritmetice asimptotice la lărgimea de bandă a memoriei în cazul relevant al suprapunerii transferurilor la memorie care are loc la FPS-164 (vezi §1.3.6 şi ecuaţia (1.20)). Lărgimea de bandă a accesului la memorie, r_∞^m, este, pentru plăcile MAX de un cuvînt într-o perioadă, iar r_∞^a de 62 operaţii aritmetice într-o perioadă, de aici

$$f_{1/2} = 31 \quad (2.20)$$

Performanţa medie poate fi estimată din (1.22b) ca

$$\bar{r}_\infty = r_\infty^a \text{ Knee } (0,5/f_{1/2}) \quad x = f/f_{1/2} \quad (2.21a)$$

de unde

$$x = 1,9 \quad \bar{r}_\infty = 0,97 r_\infty^a \quad (2.21b)$$

Astfel, găsim că atunci cînd condiţiile de reutilizare în spaţiu şi timp sînt satisfăcute, este posibilă atingerea unei performanţe în limita de 3% faţă de cea maximă.

Plăcile MAX pot executa numai un număr limitat de instrucţiuni de tipul celei din tab. 2.10. Plăcile MAX şi registrele lor sînt mapate în memorie, în ultimul Mw din cei 16 Mw de memorie adresabilă, aşa cum s-a arătat în fig. 2.49. Cu alte cuvinte, plăcile sînt folosite prin scrieri şi citiri din zonele corespunzătoare ale ultimului Mw de spaţiu de adrese. Acest

Tabelul 2.10 **Instrucţiunile care pot fi executate de o placă MAX şi performanţa maximă în Mflop/s pentru 1 şi 15 plăci. Aceleaşi valori se obţin atît pentru operaţii cu numere reale, cît şi complexe Operaţiile vectoriale totale folosesc J(I) = I.**

Nume	linie de program FORTRAN	plăci MAX	
		1	15
Produs scalar	S = S + A(I) B(J(I))	22	341
Produs scalar complex	S = S + A(I) × B(I)	22	341
VSMA	A(J(I)) = S × B(J(I)) + C(I)	11	167
VMSA	A(J(I)) = B(J(I)) × C(I) + S	11	167
Înmulţire vectorială	A(J(I)) = B(I) × C(J(I))	5,5	83
Adunare vectorială	A(J)(I)) = B(I)+ C(J(I))	5,5	83

spaţiu este împărţit în 16 zone MAX de 64 Kw fiecare. Primele 15 corespund celor 15 plăci MAX, în timp ce ultimul este segmentul de emisie care lucrează simultan cu toate plăcile. Primii 32 Kw de memorie reprezintă

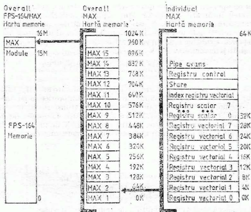


Fig. 2.49 Organizarea memoriei plăcilor MAX în spaţiul de adresare de 16 Mw al unui FPS-164.

8 registre vectoriale a 4 K elemente fiecare. Prima implementare MAX limitează lungimea vectorului la 2 K elemente. Urmează 8 registre scalare şi registrele vectoriale de index. Placa MAX funcţionează prin plasarea cuvintelor corespunzătoare în secţiunea de „avansare în pipe".

Natura programelor disponibile pe FPS-164/MAX poate fi înţeleasă din următorul cod FORTRAN care realizează înmulţirea matricială discutată anterior :

```
CALL SYS$AVAILMAX(NUMMAX)
MAXVEC=8*NUMMAX + 4
NUMVEC=MAXVEC
DO 10 I=1, N, MAXVEC
NUMVEC = MIN(NUMVEC, N-I+1)                                    (2.22)
IF(NUMVEC.LE.0) GOTO 10
CALL (PLOADD(A(I, 1), N, 1, NUMVEC, ITMA, 1, IERR)
DO 20 J = 1,N
CALL PDOT(B(1, J), 1, N, C(I, J), 1, NUMVEC, ITMA, 1, 0, IERR)
20 CONTINUE
10 CONTINUE
```

Performanța maximă se obține prin folosirea mai multor registre vectoriale. Prima instrucțiune permite obținerea în NUMMAX a numărului de plăci MAX instalate în sistem .Deoarece fiecare conține 8 registre vectoriale, iar calculatorul gazdă FPS-164 are 4 registre vectoriale, NUMVEC va fi numărul registrelor vectoriale. Bucla DO-1 încarcă NUMVEC linii ale matricei A în registrele vectoriale, pregătind acumularea a NUMVEC produse interioare. CALL PDOT formează produsele interioare ale liniei J, proces repetat pentru toate liniile de către bucla DO-J.

2.5.9. IBM ICAP și sistemul Cornell

Atît IBM cît și Universitatea Cornell dezvoltă sisteme MIMD construite prin conectarea împreună a mai multor procesoare FPS. Masivele de procesoare slab conectate IBM (ICAP), produsul ideilor lui Enrico Clementi, este instalat la laboratorul IBM Kingston. În 1985 s-a instalat o configurație similară la IBM Scientific Center din Roma, cu scopul realizării unui prim nucleu de calcul al noului „European Center for Scientific and Engineering Computing" (ECSEC).

În fig. 2.50 este reprezentat schematic IBM ICAP, 10 calculatoare FPS-164, fiecare cu 4 MB de memorie centrală, sînt conectate prin canale de 2—3 MB/s la calculatoare gazdă IBM (Berney 1984). 7 sînt legate la un IBM 4381 și sînt dedicate calculelor propriu-zise, iar 3 conectate la un IBM 4341 sînt folosite pentru dezvoltarea programelor, deși toate 10 pot fi conectate la IBM 4381 pentru a realiza un sistem cu performanța teoretică maximă de 110 Mflop/s. Performanța obținută la rezolvarea unor probleme de chimie, în cazul configurației cu 10 FPS-164, este aproximativ aceeași cu cea de la CRAY-1S, de aproximativ 60 Mflop/s (Clementi et al 1984). Îmbunătățiri posibile ale configurației inițiale constau în adăugarea a două plăci MAX celor 10 FPS-164, ceea ce ar permite atingerea a 550 Mflop/s. Dacă fiecărui FPS 164 i s-ar adăuga 15 plăci MAX, performanța ar putea atinge 3,4 Gflop/s.

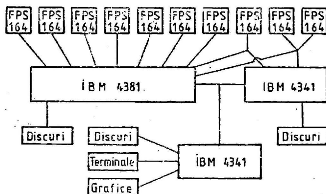


Fig. 2.50 O schemă simplificată a masivului de procesoare slab interconectate IBM (ICAP) de la Kingston, unde 10 FPS 164 comunică printr-un calculator gazdă IBM.

Acest sistem de calcul este descris ca un masiv slab conectat, deoarece în configurația inițială nu există nici o legătură directă între elementele de calcul (FPS-164) și deoarece legătura cu calculatorul gazdă se face prin canale lente. În consecință, numai problemele cu o granularitate foarte mare a paralelismului pot fi rezolvate eficient. Cu alte cuvinte, fiecare FPS-164 trebuie să execute un număr mare de operații înainte de a transfera rezultatele calculatorului gazdă prin canalele lente, sau prin calculatorul gazdă către alte FPS-164.

Un proiect similar a fost dezvoltat pe parcursul cîtorva ani la Cornell University's Theory Center, sub conducerea Prof. Kenneth Wilson. Inițial, acesta constă din 8 procesoare FPS-100, conectate printr-o magistrală de 24 MB/s. Proiectul este continuat la Cornell Advanced Scientific Computing Center, prin subvenții din partea NSF, IBM și FPS. Se afirmă că pot fi conectate pînă la 4000 plăci MAX, care ar realiza 40 Gflop/s.

Pentru a cuantifica timpii de sincronizare și comunicare la ICAP, s-au efectuat măsurători ale parametrilor ($\hat{r}_\infty$, $\hat{s}_{1/2}$, $f_{1/2}$), discutate în §1.3.6 partea (iv). Testul s-a făcut fie cu utilizarea canalelor, fie cu utilizarea magistralei FPS BUS, ceea ce a condus la următoarele ecuații (Hockney 1987d)

$$t_{canal} = 2500 + 4777\ p + (1{,}87\ m/p) + t_a(p)s\ \mu s \tag{2.23a}$$

$$t_{BUS} = 18926 + 8181p + 0{,}191\ m + t_a(p)s\ \mu s \tag{2.23b}$$

unde m este numărul cuvintelor transferate în I/E, iar s numărul operațiilor în virgulă mobilă din fiecare segment de lucru (vezi §1.3.6, partea (iv)). Primii doi termeni ai ecuației (2.23) corespund timpului necesar sincronizării, al treilea se referă la comunicații, iar ultimul este cel consumat de operații. Faptul că termenul corespunzător comunicației este invers proporțional cu numărul procesoarelor, p, în cazul folosirii canalelor, arată că acestea, deși lente, lucrează în paralel. Pe de altă parte, vedem că în cazul folosirii magistralei FPS BUS, timpul pentru comunicații nu depinde de p, ceea ce arată că aceasta, deși mult mai rapidă, lucrează serial.

Pentru a compara cele două variante, egalăm formulele (2.23a) cu (2.23b), obținînd linia de performanță egală (EPL):

$$m = p\frac{(16426 + 3404p)}{(1{,}87 - 0{,}191p)} \tag{2.24}$$

Această relație este trasată în planul fazelor (p, m) în fig. 2.51. Pentru un număr de procesoare p și un număr m de cuvinte, se obține un punct în plan. Poziția lui determină dacă trebuie folosită comunicația prin magistrală sau canal. Pentru p = 9,78 (linia întreruptă) se obține infinit, ceea ce arată că canalele vor fi întotdeauna mai rapide dacă se folosesc cel puțin 10 procesoare. Motivul este faptul că overhead-ul de inițializare și de comunicații este mai mic în cazul canalelor, decît în cazul magistralei, iar viteza asimptotică este mai mare dacă există cel puțin 10 calculatoare care lucrează în paralel. Cu alte cuvinte, mai mult de 10 canale de 2—3 MB care lucrează în paralel sînt mai rapide decît o magistrală de 24 MB care lucrează serial. În fig. 2.51 este reprezentată și curba ce corespund e

Acest sistem de calcul este descris ca un masiv slab conectat, deoarece în configurația inițială nu există nici o legătură directă între elementele de calcul (FPS-164) și deoarece legătura cu calculatorul gazdă se face prin canale lente. În consecință, numai problemele cu o granularitate foarte mare a paralelismului pot fi rezolvate eficient. Cu alte cuvinte, fiecare FPS-164 trebuie să execute un număr mare de operații înainte de a transfera rezultatele calculatorului gazdă prin canalele lente, sau prin calculatorul gazdă către alte FPS-164.

Un proiect similar a fost dezvoltat pe parcursul cîtorva ani la Cornell University's Theory Center, sub conducerea Prof. Kenneth Wilson. Inițial, acesta constă din 8 procesoare FPS-100, conectate printr-o magistrală de 24 MB/s. Proiectul este continuat la Cornell Advanced Scientific Computing Center, prin subvenții din partea NSF, IBM și FPS. Se afirmă că pot fi conectate pînă la 4000 plăci MAX, care ar realiza 40 Gflop/s.

Pentru a cuantifica timpii de sincronizare și comunicare la ICAP, s-au efectuat măsurători ale parametrilor ($\hat{r}_\infty$, $\hat{s}_{1/2}$, $f_{1/2}$), discutate în §1.3.6 partea (iv). Testul s-a făcut fie cu utilizarea canalelor, fie cu utilizarea magistralei FPS BUS, ceea ce a condus la următoarele ecuații (Hockney 1987d)

$$t_{canal} = 2500 + 4777\ p + (1{,}87\ m/p) + t_a(p)s\ \mu s \tag{2.23a}$$

$$t_{BUS} = 18926 + 8181p + 0{,}191\ m + t_a(p)s\ \mu s \tag{2.23b}$$

unde m este numărul cuvintelor transferate în I/E, iar s numărul operațiilor în virgulă mobilă din fiecare segment de lucru (vezi §1.3.6, partea (iv)). Primii doi termeni ai ecuației (2.23) corespund timpului necesar sincronizării, al treilea se referă la comunicații, iar ultimul este cel consumat de operații. Faptul că termenul corespunzător comunicației este invers proporțional cu numărul procesoarelor, p, în cazul folosirii canalelor, arată că acestea, deși lente, lucrează în paralel. Pe de altă parte, vedem că în cazul folosirii magistralei FPS BUS, timpul pentru comunicații nu depinde de p, ceea ce arată că aceasta, deși mult mai rapidă, lucrează serial.

Pentru a compara cele două variante, egalăm formulele (2.23a) cu (2.23b), obținînd linia de performanță egală (EPL):

$$m = p\frac{(16426 + 3404p)}{(1{,}87 - 0{,}191p)} \tag{2.24}$$

Această relație este trasată în planul fazelor (p, m) în fig. 2.51. Pentru un număr de procesoare p și un număr m de cuvinte, se obține un punct în plan. Poziția lui determină dacă trebuie folosită comunicația prin magistrală sau canal. Pentru p = 9,78 (linia întreruptă) se obține infinit, ceea ce arată că canalele vor fi întotdeauna mai rapide dacă se folosesc cel puțin 10 procesoare. Motivul este faptul că overhead-ul de inițializare și de comunicații este mai mic în cazul canalelor, decît în cazul magistralei, iar viteza asimptotică este mai mare dacă există cel puțin 10 calculatoare care lucrează în paralel. Cu alte cuvinte, mai mult de 10 canale de 2—3 MB care lucrează în paralel sînt mai rapide decît o magistrală de 24 MB care lucrează serial. În fig. 2.51 este reprezentată și curba ce corespund e

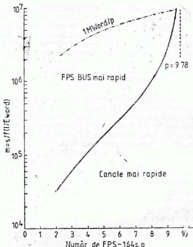


Fig. 2.51 Diagrama de fază prin care se compară utilizarea FPS BUS cu cea a canalelor. Pentru orice număr ales de procesoare, p, fie FPS BUS, fie canalele sînt mai rapide funcție de numărul cuvintelor transferate m. Pentru mai mult de 10 procesoare, întotdeauna canalele sînt mai rapide.

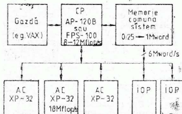


Fig. 2.52 Arhitectura seriei de calculatoare FPS-5000.

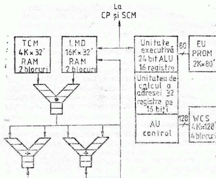


Fig. 2.53 Arhitectura internă a coprocesorului aritmetic FPS × P-32.

la 10^{+55} la 10^{+153}, cu o precizie de 28 biţi. Circuitele WEITEK au adoptat standardul IEEE 754, de reprezentare pe 32 biţi (IEEE 1983), care are un domeniu de reprezentare a valorilor absolute mult mai redus, de la 10^{-38} la 10^{-38}, dar o mantisă mai precisă, echivalentă cu 33 biţi.

Arhitectura co-procesorului XP-32 (fig. 2.53) este similară cu cea de la AP-120B, dar diferă la detalii (FPS 1984c). Un multiplicator pipeline cu 5 etaje şi două sumatoare cu 5 etaje sînt conectate prin căi de date multiple la o *memorie principală locală* (LMD) şi o *memorie pentru tabele cu coeficienţi* (TCM). LMD are capacitatea de 16 K cuvinte pe 32 biţi, organizate în două blocuri, iar TCM are 4 K cuvinte pe 32 biţi organizate de asemenea în două blocuri. Controlul general este realizat de *unitatea executivă* (EU) care poate lucra simultan cu *unitatea aritmetică* (AU), asigurînd astfel efectuarea în paralel a operaţiilor de I/E şi calcul al adreselor. EU execută toate transferurile de programe şi date între AC şi CP sau SCM. AU execută operaţii aritmetice numai cu date din TCM şi LMD. Microprogramele pentru EU rezidă în EU PROM, care conţine 2 K microinstrucţiuni pe 80 biţi. Similar, microprogramele pentru AU se află într-o memorie (*writable control store*) de 4 K microinstrucţiuni pe 128 biţi, organizată în 4 blocuri.

Între co-procesoarele aritmetice nu există nici o conexiune directă. Ele pot numai prelua date din SCM şi returna rezultate în SCM, astfel că pot comunica numai prin partajarea datelor în SCM.SCM lucrează ca memorie centrală a procesorului de control. În cazul procesorului de control AP-120B, perioada ceasului este de 167 ns. În cazul lui FPS-100, memoria lucrează cu o perioadă de 250 ns, deci este mai lentă. Co-procesoarele aritmetice pot avea acces direct (DMA) la memoria SCM, luînd locul lui CP, conform unei scheme de prioritate. În cazul accesării de către

ACS, SCM poate fie să scrie, fie să citească un cuvînt într-o perioadă de ceas (dar nu ambele în același timp), obținîndu-se o lărgime de bandă a accesului la SCM de fie 6 Mw/s (24 MB/s), fie 4 Mw/s (16 MB/s). Memoria este astfel organizată încît fiecare XP-32 individuală să folosească numai jumătate din ea, permițînd astfel realizarea accesului simultan a două ACS, înainte ca accesul la memorie să devină restrictiv. În fiecare ciclu de memorie se limitează numărul cererilor de acces la memorie a oricărui ACS.

FPS-5000 poate fi programat în întregime să folosească apeluri la subprogramele de bibliotecă într-un subset FORTRAN 77, denumit CP FORTRAN. În cele mai multe situații documentația de firmă furnizează formule de evaluare a timpului de execuție, de unde pot fi deduse r_∞ și $n_{1/2}$ (FPS 1984a, b, 1985a). Prezentăm în continuare cîteva selecții care indică posibilitățile sistemului.

(i) *Rutinele de interfațare cu gazda*

Următoarele rutine se execută pe calculatorul gazdă, încarcă programe și date în FPS-5000 și lansează în execuție programul CP FORTRAN, pe CP

CPOPEN Deschide un fișier program CP FORTRAN.

CPLOAD Încarcă un fișier program CP FORTRAN de la gazdă în CP.

CPRUN Lansează în execuție programul CP FORTRAN pe CP.

EXPUT Începe transferul datelor de la gazdă la FPS-5000.

EXGET Începe transferul datelor de la FPS-5000 la gazdă.

APWAIT Așteaptă încheierea transferului de date și a programului CP.

APWD Așteaptă încheierea transferului de date.

APWR Așteaptă oprirea programului CP.

(ii) *Sincronizarea ACS de către CP*

Următoarele rutine se execută pe CP și controlează ACS:

XPSEL Selectează XP-32 pentru XPWAIT.

XPRUN Începe execuția programului pe XP-32 selectat.

XPWAIT Așteaptă pentru ca XP-32 selectat să-și încheie lucrul.

XPSTAT Obține starea XP-32.

(iii) *Transfer de date cu SCM*

Următoarele rutine se execută pe XP-32, transferînd date înainte sau după terminarea calculelor:

XPDMAR Transferă date între SCM și LMD.

XTMDMA Transferă date între SCM și TCM.

$r_\infty = 2$ Mop/s.

XPISNC Așteaptă ca transferul (sau operațiile aritmetice) să se încheie.

(iv) *Operații aritmetice în XP-32*

Următoarele rutine XPMLIB se execută pe XP-32, realizînd operații aritmetice cu date din LMD:

ZVMUL (IA, IB, IC, N) Înmulțire vectorială element cu element A*B; cele N elemente produse se depun în C ($r_\infty = 4$ Mflop/s, $n_{1/2} = 33$).

ZVDIV (IA, IB, IC, N) Împărțire vectorială element cu element ($r_\infty = 0,5$ Mflop/s, $n_{1/2} = 9$).

ZVSASM (IA, IB, ID, IC, N) Triada vectorială, adunare scalar-vector, înmulțire scalar-vector: $C = (A+b)*d$ ($r_\infty = 12$Mflop/s, $n_{1/2}=56$).

ZVASM (IA, IB, ID, IC, N) $C = (A+B)*d$ (r_∞=8 Mflop/s, $n_{1/2}$= = 37).

ZVAM (IA, IB, ID, IC, N) $C = (A+B)*D$ (r_∞=6 Mflop/s, $n_{1/2}$= = 28).

Cifrele furnizate mai sus corespund unor operaţii diadice sau triadice executate de un singur XP-32, neluîndu-se în considerare timpul necesar sincronizării multiplelor ACS ale unui FPS-5000 (performanţa MIMD) sau timpul de transfer al datelor din SCM la LMD înaintea execuţiei calculelor. Aceste valori au fost măsurate separat de Curington şi Hockney (1986) şi interpretate în termenii parametrilor $n_{1/2}$, $s_{1/2}$ şi $f_{1/2}$ (§1.3.6). În tab. 2.11 şi 2.12 se prezintă aceste rezultate.

Calculatorul FPS-5000A, folosit pentru măsurători, constă dintr-un procesor de control şi fie unul, fie două co-procesoare aritmetice XP-32. Măsurătorile efectuate numai cu CP (tab. 2.11) nu implică sincronizare şi sînt de valoarea lui $n_{1/2}$ în timp ce cele produse pe multiprocesoare includ timpul de sincronizare şi sînt de valoarea lui $s_{1/2}$. Valorile ultimului indică numărul minim de operaţii aritmetice care trebuie împărţite între coprocesoarele XP-32. Atît CP cît şi ACS lucrează cu frecvenţa de 6 MHz şi au unităţi pipeline cu performanţă maximă de 6 Mflop/s pentru operaţii diadice, care folosesc numai un pipeline, sau 12 Mflop/s pentru operaţiile triadice care folosesc două unităţi pipeline. Prin urmare, performanţa maximă pentru configuraţia maximă de un procesor de control şi două XP-32 este de 18 Mflop/s pentru diade şi 36 Mflop/s pentru triade. În celelalte cazuri, lărgimea de bandă a accesului la memorie inadecvate

Tabelul 2.11. **Măsurători (r_∞, $n_{1/2}$) pentru un multiprocesor FPS-5320 A care are un procesor de control (CP) şi unul sau două coprocesoare XP-32, care lucrează cu date din memoriile lor locale.**

Operaţie	Configuraţie	r_∞ (Mflop/s)	n_{12} sau $s_{1/2}$
Diadă	numai CP	1,5	14*
A = B × C	un XP-32	4,0	470
VMUL sau ZVMUL	două XP-32	8,0	1320
	CP + două XP-32	9,2	1545
Triadă	numai CP	3,9	40*
A = (B + s) xc	un XP − 32	12,0	1490
VSASM sau ZVSASM	două XP-32	24,0	4200
	CP + două XP-32	27,7	4820

Tabelul 2.12 **Valori ale performanţei maxime, r_∞ şi $f_{1/2}$ pentru un coprocesor aritmetic XP-32 care execută operaţii ZVASASM cu date în memoria comună.**

Cazul	r_∞ (Mflop/s)	$f_{1/2}$
I/E secvenţială	12,5	4,2
I/ E suprapuse	12,6	2,2

împiedică atingerea valorilor maxime, deși, din acest punct de vedere, XP-32 suferă mult mai puțin decît CP.

Măsurătorile din tab. 2.11 corespund operațiilor executate cu date din memoriile locale (LMD) ale XP-32, care se presupun deja încărcate. În mod obișnuit, datele se află în SCM și vor trebui transferate în LMD înaintea efectuării calculelor. Prin urmare, viteza de calcul totală va depinde în mod critic de numărul operațiilor aritmetice executate în XP-32 pentru un transfer între SCM și LMD (variabila f definită în §1.3.6)și poate fi caracterizată de parametrul $f_{1/2}$. FPS-5000 este un instrument ideal pentru studiul acestei dependente, deoarece f poate fi modificat cu ușurință, obținîndu-se și valoarea necesară atingerii a jumătate din performanța maximă. Parametrul $f_{1/2}$ este furnizat în tab. 2.12. Valoarea obținută de 12 Mflop/s corespunde operației ZVSASM. Se consideră două cazuri, primul cînd transferurile de I/E între SCM și LMD au loc secvențial cu operațiile aritmetice și al doilea, cînd operațiile de I/E au loc simultan cu cele aritmetice (se suprapun). Se găsește că efectul suprapunerii este înjumătățirea lui $f_{1/2}$.

3.

MULTIPROCESOARE ŞI MASIVE DE PROCESOARE

3.1. Limitele conceptului pipeline

Paralelismul poate fi introdus în structura fizică a calculatorului numai prin două tehnici, multiplicare şi pipelining; conceptul pipeline poate fi interpretat ca o multiplicare posibilă prin secvenţă, deoarece fiecare componentă urmează alteia în timp. Operaţiile pipeline se execută prin suprapunerea operaţiilor lor componente mai simple, folosindu-se hardware paralel. Părţile componente ale secvenţei de operaţii sînt prelucrate în pipeline în cursul instanţelor sucvesive de timp. În acest mod, o singură operaţie va partaja unitatea pipeline cu un număr de alte operaţii în timp ce execuţia ei avansează prin diversele etaje.

Diferenţa fundamentală dintre pipelining şi multiplicarea spaţială este că operaţiile componente paralel ale unui pipeline vor executa probabil task-uri diferite care, cînd se execută în secvenţă, realizează operaţia cerută. Evident există o limită a paralelismului disponibil prin descompunerea unei operaţii în sub-task-uri, în afară de cazul cînd operaţiile sînt extrem de complexe. Deşi programele complete sînt complexe, iar utilizarea unui pipeline de task-uri concurente poate fi foarte atractivă ca stil de programare, astfel de unităţi pipeline mai sînt foarte dependente de aplicaţie. Astfel, în cazul general, conceptul pipelining poate asigura un grad limitat de paralelism, prin exploatarea operaţiilor complexe comune, cum sînt cele de calcul aritmetic în virgulă mobilă.

Totuşi conceptul pipelining reprezintă forma cea mai atractivă de paralelism disponibil, deoarece nu creează aceleaşi probleme de comunicaţie întîlnite la multiplicarea spaţială. Un pipeline este proiectat să reflecte circulaţia naturală a datelor operaţiei în curs de execuţie, în timp ce multiplicarea spaţială va utiliza fie o reţea fixă, fie o reţea programabilă. În primul caz, reţeaua nu reflectă în mod necesar circulaţia datelor prelucrate; în al doilea caz, mai general, costurile sînt mari.

Deşi este atractiv, singur conceptul pipelining nu va asigura atingerea scopului unei puteri de calcul nelimitate, care a justificat folosirea paralelismului în sistemele de calcul. Acesta se va realiza prin utilizarea

multiplicării spaţiale, sau într-adevăr printr-o combinaţie între pipelining şi multiplicarea spaţială. Această teză pare a fi deja demonstrată, lucru evidenţiat de evoluţia calculatoarelor pipeline descrise în capitolul 2. CRAY X-MP a evoluat prin multiplicarea unei secţiuni de calcul identică funcţional cu cea de la predecesorul CRAY-1, dar mai rapidă. Un alt exemplu bun este oferit de CDC CYBER 205, evoluat din CDC STAR prin multiplicarea unităţilor pipeline reproiectate. Într-adevăr, această arhitectură are un viitor mare reprezentat de ETA[10], unde o implementare VLSI a unui CYBER 205 cu două unităţi pipeline va fi multiplicată de pînă la 8 ori pentru atingerea performanţei maxime de 10 Gflop/s.

Aceste arhitecturi ce se bazează pe multiplicarea limitată a unor procesoare foarte rapide sînt practic foarte costisitoare. Ele folosesc orologii rapide, care impun folosirea unor circuite de viteză mare şi care disipă multă energie, cum sînt cele în tehnologie ECL, care nu permit o densitate prea mare. Mai mult, pentru atingerea unor viteze înalte, circuitele trebuie să fie atît de apropiate cît este fizic posibil. În acest mod se evită limitarea frecvenţei orologiilor prin timpii de propagare ai semnalelor. În schimb se exacerbează problemele eliminării puterii disipate de aceste circuite fierbinţi. ETA[10] reprezintă o excepţie de la folosirea circuitelor ECL la supercalculatoare, fiind implementat cu circuite CMOS de mare densitate. Pentru a se obţine în tehnologie VLSI viteze mari de comutare, sînt necesare nivele de tensiune scăzute şi temperaturi de lucru foarte scăzute (temperatura hidrogenului lichid).

Deşi ETA[10] foloseşte circuite VLSI, nu exploatează în întregime posibilităţile de miniaturizare oferite de tehnologie. Sînt necesare multe circuite diferite, proiectate pe baza tehnicii de proiectare a masivelor de porţi. Alternativa circuitelor VLSI de masă implică o multiplicare masivă în sistem. Recent (circa 1985) au devenit disponibile componente adecvate pentru multiplicarea în acest grad. Acest capitol explorează alternativele replicării în arhitectura calculatorului; aspectele tehnologice sînt tratate ulterior, în capitolul 6.

3.2. Alternativa multiplicării

3.2.1. O problemă de scalare

În sistemele cu multiplicare, aspectele cele mai importante sînt legate de scalare; în timp ce un număr mic de procesoare pot comunica eficient prin magistrale sau memorii comune, în cazul unor numere mai mari natura inerent secvenţială a acestor metode produce strangulări. De aceea, proiectantul este forţat să considere sisteme în care datele sînt comutate între procesoare. Aceasta se poate realiza fie cu o reţea fixă, unde distanţele cresc cu numărul de procesoare, fie cu o reţea programabilă, unde costurile urmează o lege pătratică. (Ultima soluţie poate ridica probleme considerabile privind conductoarele electrice).

Fig. 3.1 prezintă schematic un sistem multiplicat spaţial general. Este o încercare de a distila caracteristicile comune unei varietăţi largi

de astfel de sisteme şi de aceea nu reflectă nici un sistem particular. Totuşi, ilustrează aspectele mai importante ale unor astfel de arhitecturi. Acestea sint : o memorie paralelă de unde pot fi accesate simultan mai multe cuvinte ; un masiv de procesoare ; şi circuitele de comutare, importante peste tot. Comutatorul dintre procesoare asigură un set de conexiuni între porturile procesoarelor, care pot realiza un set redus şi fix de permutări, sau pot realiza setul complet de n! permutări. Comutatorul dintre procesor-memorie asigură căile de date între diversele blocuri de memorie şi procesoare. Pentru acest comutator există două alternative : una permite numai permutarea identică, iar cealaltă permite unui procesor accesul la toate blocurile de memorie. Trebuie să observăm că nu este de dorit să avem, în acest ultim caz, un număr de procesoare egal cu cel al blocurilor de memorie.

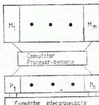


Fig. 3.1 Diagramă ce ilustrează un sistem generic cu multiplicare spaţială, unde P_1 la P_n sint procesoare, iar M_1 la M_m sint modulele de memorie.

Diferenţele între arhitecturile paralele diverse ce se intilnesc în literatură pot fi cuantificate cu următorii parametrii :

(a) prima cantitate este dimensionarea masivului şi puterea procesorului individual, produsul acestora determinind pînă la un punct puterea întregului sistem ;

(b) complexitatea reţelelor de conectare, care va determina flexibilitatea sistemului şi, de aici, dacă puterea obţinută prin multiplicare poate fi folosită de o clasă mare de probleme ;

(c) distribuţia controlului sistemului, adică dacă întregul masiv este condus de un procesor central, sau dacă fiecare procesor are propriul său controler ;

(d) forma de control a sistemului, care poate fi dedusă dintr-o secvenţă, predefinită de instrucţiuni, sau o structură de control mai adecvată stilurilor de programare declarative, ca modelul data flow sau reducţionist.

Trebuie să observăm că în acest model procesoarele pot să nu aibă o structură atomică simplă, dar pot conţine multiplicare sau concurenţă, ca în cazul structurilor pipeline multiplicate amintite anterior.

3.2.2. Organizarea controlului

Din toate problemele ridicate anterior, probabil controlul este cel care diferenţiază arhitecturile paralele. Cu siguranţă acesta este adevărul, cind se consideră forma de ansamblu de control (cu flux de instrucţiuni, reducţionist, data flow etc.), iar alegerea modului de control al următoarei generaţii de calculatoare, a cincea generaţie (Uehida 1983) este însoţită de controverse.

Atît stategia data flow (Dennis 1980, Chamber et al 1984), cit şi cea reducţionistă (Turner 1982, Chamber et al 1984) pot fi utile pentru degrevarea programatorului de sarcina secvenţierii explicite a instrucţiuni-

lor. Efectul asupra ingineriei programării este profund (vezi cap. 4), asociat cu un cadru pentru strategia de lucru a mașinii. În aceste limbaje funcționale și logice, secvența de instrucțiuni se stabilește prin decompoziția descrierii problemei, mai degrabă decît pe baza unui algoritm.

(i) *Data flow*

În cadrul modelului data flow, o instrucțiune nu se execută sub influența unui numărător de program, ci dacă și numai dacă toți operanzii ei sînt disponibili. Prin urmare, un program data flow poate fi interpretat ca un graf orientat, de-a lungul căruia circulă elemente de date (data token), ieșirea unei operații fiind conectată cu un arc al grafului la operațiile care îi consumă rezultatul. Într-o mașină fizică, instrucțiunile ar fi reprezentate în general ca pachete ce conțin cîmpuri pentru operații, operanzi (date sau referințe) și identificatori care dau semnificații acestor informații. Acest ultim cîmp este necesar, deoarece starea mașinii nu mai poate asigura contextul de unde poate fi dedusă interpretarea datelor. În cursul execuției identificatorii sînt comparați cu pachetele rezultate, ce conțin cîmpuri de identificatori, și date și cînd toți operanzii unei instrucțiuni sînt disponibili, instrucțiunea este plasată într-o coadă de așteptare pentru execuție imediată.

Pentru exemplificare să considerăm programul și modul de execuție pentru expresia :

$$(A+B)*(C+D)$$

Programul, care conține un număr de pachete instrucțiuni este „încărcat" în sistem prin injectarea acestor pachete într-o memorie pentru programe. Execuția programului va fi inițiată prin injectarea pachetelor de date, care conțin valorile pentru A, B, C și D. O implementare a variabilelor A și D este prin atribuirea unor identificatori unici, memorați în pachetele instrucțiuni. Datele vor trebui identificate în același mod. O unitate de comparare va compara identificatorii datelor care circulă prin sistem cu cei ai instrucțiunilor. De exemplu, cele două operații de adunare vor atrage datele asociate, iar cînd sînt gata pentru execuție, ele se pot realiza în paralel. Calculatoarele data flow pot folosi conceptele de pipelining și multiplicare.

Conceptul pipelining poate fi introdus în circulația pachetelor de date (secvența de operații) prin sistem. Multiplicarea poate fi realizată prin partajarea pachetelor de instrucțiuni între procesoare. Dacă se exploatează această ultimă formă de paralelism, sînt necesare mijloace echitabile de partajare a pachetelor program și a identificatorilor asociați între procesoare.

Numai cînd s-au terminat aceste două operații de adunare, generîndu-se valori pentru sub-expresiile din paranteze, va putea fi executată și înmulțirea. Se poate vedea că strategia de execuție este determinată de date și începe de la nivelul cel mai interior al unei expresii, desfășurîndu-se spre exterior. Evident, într-un program real, graful sau programul vor fi mult mai complexe decît acest exemplu simplu. Oricum, acest exemplu este suficient pentru ilustrarea noțiunii de paralelism asincron ce este controlat în întregime de un mecanism determinat de date. Deoarece toate

dependențele de date au fost rezolvate prin graful de descriere al programului, nu sînt necesare declarații paralele explicite care să faciliteze execuția programelor data flow pe procesoare multiple. Programele trebuie descompuse în modalități care să permită extragerea paralelismului. De exemplu, o simplă recursie de listă va produce un algoritm secvențial, în timp ce o împărțire recursivă a listei, ce ar exprima funcția la ambele jumătăți, va genera un algoritm ce conține paralelism. Înjumătățirea recursivă este la baza multor algoritmi, de exemplu sortarea quick și este o expresie clasică pentru generarea paralelismului implicit.

O mașină data flow ce se bazează pe conceptul pachetelor de instrucțiuni marcate cu identificatori va avea unul sau mai multe procesoare constituite din următoarele unități:

(a) una sau mai multe memorii pentru date;

(b) una sau mai multe unități de execuție;

(c) o memorie program, unde se află graful program;

(d) o unitate de comparare, care elimină informațiile neprelucrate.

O astfel de mașină a fost construită de un grup de cercetători de la Universitatea Manchester (Gurd și Watson 1980, și Watson și Gurd 1982). Pentru lecturi ulterioare privind conceptul data flow recomandăm lucrările lui Chamber et al (1984) și Hwang și Briggs (1984).

(ii) *Reducerea*

Reducerea ca metodă de control a calculatorului poate produce, de asemenea, paralelism fără control explicit. Reducerea se bazează pe matematica funcțiilor și calculul lambda, iar prin folosirea acestui formalism programele pot fi considerate ca șiruri de expresii sau arbori. De exemplu, un program poate fi reprezentat de expresia următoare, fie ca șir, fie structurat ca arbore, cu operatorul „*" ca rădăcină și doi sub-arbori ce conțin operatori „+" și operanzii A și B, respectiv C și D.

$$((A+B)*(C+D))$$

În timp ce la mașinile data flow execuția este comandată de date, în cazul unei strategii reducționiste execuția se face la cerere (demand driven). Astfel, dacă acest program a fost introdus în sistem sau activat printr-o cerere a rezultatului, emisă de un program mai mare, va avea loc o serie de rescrieri, care reduc această expresie la operațiile componente. O rescriere este procedura de reducere a unei expresii sau arbore, de execuție a operației dacă sînt cunoscute frunzele, sau de activare a subexpresiilor sau sub-arborilor dacă nu sînt cunoscute. Probabil termenul de „reducere" induce în eroare, deoarece operațiile inițiale ale secvenței vor genera mai multe programe de executat odată cu acționarea unor noi sub-arbori sau sub-expresii.

Desigur fiecare arbore poate fi distribuit pe hardware concurent, pentru o evaluare în paralel. La o etapă ulterioară, programul va fi redus la operațiile componente, care pot fi reprogramate similar ca la sistemele data flow, prin pachete marcate cu identificatori. Se poate vedea că reducerea evaluează expresiile din exterior spre interior.

lor. Efectul asupra ingineriei programării este profund (vezi cap. 4), asociat cu un cadru pentru strategia de lucru a mașinii. În aceste limbaje funcționale și logice, secvența de instrucțiuni se stabilește prin decompoziția descrierii problemei, mai degrabă decît pe baza unui algoritm.

(i) *Data flow*

În cadrul modelului data flow, o instrucțiune nu se execută sub influența unui numărător de program, ci dacă și numai dacă toți operanzii ei sînt disponibili. Prin urmare, un program data flow poate fi interpretat ca un graf orientat, de-a lungul căruia circulă elemente de date (data token), ieșirea unei operații fiind conectată cu un arc al grafului la operațiile care îi consumă rezultatul. Într-o mașină fizică, instrucțiunile ar fi reprezentate în general ca pachete ce conțin cîmpuri pentru operații, operanzi (date sau referințe) și identificatori care dau semnificații acestor informații. Acest ultim cîmp este necesar, deoarece starea mașinii nu mai poate asigura contextul de unde poate fi dedusă interpretarea datelor. În cursul execuției identificatorii sînt comparați cu pachetele rezultate, ce conțin cîmpuri de identificatori, și date și cînd toți operanzii unei instrucțiuni sînt disponibili, instrucțiunea este plasată într-o coadă de așteptare pentru execuție imediată.

Pentru exemplificare să considerăm programul și modul de execuție pentru expresia :

$$(A+B)*(C+D)$$

Programul, care conține un număr de pachete instrucțiuni este „încărcat" în sistem prin injectarea acestor pachete într-o memorie pentru programe. Execuția programului va fi inițiată prin injectarea pachetelor de date, care conțin valorile pentru A, B, C și D. O implementare a variabilelor A și D este prin atribuirea unor identificatori unici, memorați în pachetele instrucțiuni. Datele vor trebui identificate în același mod. O unitate de comparare va compara identificatorii datelor care circulă prin sistem cu cei ai instrucțiunilor. De exemplu, cele două operații de adunare vor atrage datele asociate, iar cînd sînt gata pentru execuție, ele se pot realiza în paralel. Calculatoarele data flow pot folosi conceptele de pipelining și multiplicare.

Conceptul pipelining poate fi introdus în circulația pachetelor de date (secvența de operații) prin sistem. Multiplicarea poate fi realizată prin partajarea pachetelor de instrucțiuni între procesoare. Dacă se exploatează această ultimă formă de paralelism, sînt necesare mijloace echitabile de partajare a pachetelor program și a identificatorilor asociați între procesoare.

Numai cînd s-au terminat aceste două operații de adunare, generîndu-se valori pentru sub-expresiile din paranteze, va putea fi executată și înmulțirea. Se poate vedea că strategia de execuție este determinată de date și începe de la nivelul cel mai interior al unei expresii, desfășurîndu-se spre exterior. Evident, într-un program real, graful sau programul vor fi mult mai complexe decît acest exemplu simplu. Oricum, acest exemplu este suficient pentru ilustrarea noțiunii de paralelism asincron ce este controlat în întregime de un mecanism determinat de date. Deoarece toate

dependențele de date au fost rezolvate prin graful de descriere al programului, nu sînt necesare declarații paralele explicite care să faciliteze execuția programelor data flow pe procesoare multiple. Programele trebuie descompuse în modalități care să permită extragerea paralelismului. De exemplu, o simplă recursie de listă va produce un algoritm secvențial, în timp ce o împărțire recursivă a listei, ce ar exprima funcția la ambele jumătăți, va genera un algoritm ce conține paralelism. Înjumătățirea recursivă este la baza multor algoritmi, de exemplu sortarea quick și este o expresie clasică pentru generarea paralelismului implicit.

O mașină data flow ce se bazează pe conceptul pachetelor de instrucțiuni marcate cu identificatori va avea unul sau mai multe procesoare constituite din următoarele unități:

(a) una sau mai multe memorii pentru date;

(b) una sau mai multe unități de execuție;

(c) o memorie program, unde se află graful program;

(d) o unitate de comparare, care elimină informațiile neprelucrate.

O astfel de mașină a fost construită de un grup de cercetători de la Universitatea Manchester (Gurd și Watson 1980, și Watson și Gurd 1982). Pentru lecturi ulterioare privind conceptul data flow recomandăm lucrările lui Chamber et al (1984) și Hwang și Briggs (1984).

(ii) *Reducerea*

Reducerea ca metodă de control a calculatorului poate produce, de asemenea, paralelism fără control explicit. Reducerea se bazează pe matematica funcțiilor și calculul lambda, iar prin folosirea acestui formalism programele pot fi considerate ca șiruri de expresii sau arbori. De exemplu, un program poate fi reprezentat de expresia următoare, fie ca șir, fie structurat ca arbore, cu operatorul „*" ca rădăcină și doi sub-arbori ce conțin operatori „+" și operanzii A și B, respectiv C și D.

$$((A+B)*(C+D))$$

În timp ce la mașinile data flow execuția este comandată de date, în cazul unei strategii reducționiste execuția se face la cerere (demand driven). Astfel, dacă acest program a fost introdus în sistem sau activat printr-o cerere a rezultatului, emisă de un program mai mare, va avea loc o serie de rescrieri, care reduc această expresie la operațiile componente. O rescriere este procedura de reducere a unei expresii sau arbore, de execuție a operației dacă sînt cunoscute frunzele, sau de activare a subexpresiilor sau sub-arborilor dacă nu sînt cunoscute. Probabil termenul de „reducere" induce în eroare, deoarece operațiile inițiale ale secvenței vor genera mai multe programe de executat odată cu acționarea unor noi sub-arbori sau sub-expresii.

Desigur fiecare arbore poate fi distribuit pe hardware concurent, pentru o evaluare în paralel. La o etapă ulterioară, programul va fi redus la operațiile componente, care pot fi reprogramate similar ca la sistemele data flow, prin pachete marcate cu identificatori. Se poate vedea că reducerea evaluează expresiile din exterior spre interior.

Un grup de cercetători de la Imperial College a construit o arhitectură pentru implementarea limbajelor funcţionale prin reducere (Darlington şi Reeve 1981). Această maşină se bazează pe transputere, iar ICL a produs cîteva maşini prototip, dintre care una pentru Imperial College în 1986. O altă implementare practică a arhitecturii reducţioniste este finanţată de programul Alvey la University College, Londra (Peyton-Jones 1987 a, b). Pentru informaţii privind activitatea practică recentă legată de maşinile data flow, reducţioniste, vezi Chamber et al (1984) şi Jesshope (1987 b.)

În ciuda faptului că ambele metode de control generează paralelismul fără o comandă explicită, ele sînt ineficiente în comparaţie cu strategia de control mult mai folosită a fluxului de instrucţiuni. În ambele cazuri organizarea poate solicita un număr substanţial de operaţii. Pot fi zeci sau chiar sute de instrucţiuni executate pentru fiecare instrucţiune folositoare. Aceasta este foarte ineficient în comparaţie cu calculatoarele cu flux de instrucţiuni înalt optimizate, în uz în ultimele trei, patru decade. Totuşi, aceste arhitecturi încearcă să ridice nivelul de abstractizare al modelului de programare către unul în care calculatorul execută specificaţia unei probleme. O analogie cu această situaţie ar reprezenta-o comparaţia între programarea în limbaj de asamblare şi într-un limbaj de nivel înalt, unde ultimul mod nu trebuie comparat cu primul în termeni de eficienţă dacă nu se ia în considerare şi eficienţa programării. În acest caz, există orientări spre arhitecturi care execută limbaje de nivel înalt (Organick 1973) care au minimizat orice pierdere de eficienţă în schimbul nivelului mai înalt de abstractizare.

În acelaşi mod, deoarece cercetările continuă în domeniul sistemelor declarative, arhitecturile vor deveni mai eficiente odată cu creşterea rafinamentului implementăriilor hardware şi a tehnologiei compilatoarelor. Într-adevăr, prezentări recente ale cercetărilor în domeniul data flow efectuate la Manchester (Gurd 1987) s-au referit la comparaţii foarte favorabile ale maşinii data flow în comparaţie cu arhitecturile convenţionale. Este probabil că implementările de limbaje funcţionale să urmeze această direcţie ,deşi sînt aproximativ cu 5 ani în urma fazelor de dezvoltare ale maşinilor data flow.

O limită fundamentală a acestor arhitecturi este cea a lărgimii de bandă a comunicaţiei între procesoare, dar ea este împărtăşită de toate sistemele care folosesc multiplicarea. Această problemă creşte cu dimensiunea sistemului şi deoarece maşinile data flow şi reducţioniste solicită distribuirea programelor şi a datelor, ca şi o comunicaţie adecvată, necesarul de lărgime de bandă este mai mare şi este probabil să devină o limită în calea acestor strategii. De exemplu în cazul acestor arhitecturi, pachetele de 100 biţi sînt obişnuite, chiar pentru operaţii pe 16 biţi. În controlul cu flux de instrucţiuni, numai unul într-o pereche de operanzi trebuie transmis prin reţeaua de comunicaţie. La maşinile data flow şi reducţioniste, trebuie comunicate mai multe pachete pentru a se executa o singură operaţie utilă. Introducerea unor întîrzieri sau folosirea conceptului pipelining este un instrument util pentru combaterea complexităţilor comunicaţiei, dar pentru un sistem efectiv trebuie să existe un echilibru între procesul de comunicaţie şi cel de calcul şi, aşa cum s-a indicat, acest echilibru este favorizat faţă de comunicare cu creşterea gradului de multiplicare.

În general, cele mai eficiente sisteme cu multiplicare, fie comunică date, fie programe între procesoare, care necesită cea mai mică lărgime de bandă. De exemplu, dacă, două procesoare trebuie să stabilească o comunicație susținută, ar fi indicat ca ele să-și transmită mai degrabă cod, decît date. Desigur la mașinele SIMD este necesar să se transmită date.

(iii) *Control cu flux de instrucțiuni.*

Dacă considerăm doar arhitecturile cu flux de instrucțiuni, există largi posibilități de implementare a strategiei de control intr-un sistem cu multiplicare. De exemplu, dacă folosim un singur controler central pentru toate procesoarele, avem o mașină SIMD. „Alternativa" este ca fiecare procesor să execute propriul flux de instrucțiuni ,cu un controler automat. În acest caz avem „conform clasificării lui Flynn, o mașină MIMD. Oricum, între aceste două extreme se află un spectru larg de strategii de control.

Să luăm următorul exemplu: la un nivel al unei arhitecturi date putem găsi un masiv de procesoare sub controlul MIMD, dar fiecare „procesor" poate conține, de asemenea, un masiv de procesoare sub control SIMD. Această structură a fost propusă de mai mulți proiectanți (Pease 1977, Jesshope 1986a, b, c, Lea 1986).

Să luăm și cealaltă situație a unui singur controler central, care asigură cea mai mare parte a cuvîntului de control pentru un masiv SIMD, dar unde fiecare procesor poate furniza local informații de control, funcție de condiții locale. Cele mai multe calculatoare SIMD au la nivelul fiecărui procesor cel puțin o capacitate rudimentară de control, deși, de obicei, aceasta se reduce la un comutator on/off. La alte masive, în sarcina procesoarelor au fost date mai multe cîmpuri de control, în timp ce sincronizarea și secvențierea se realizează tot centralizat. Dacă un astfel de masiv are o capacitate semnificativă de control local, se spune că este adaptiv. În §3.5.4. se prezintă un exemplu de masiv adaptiv.

Controlul unui masiv poate consta de asemenea din mai multe nivele, această tehnică este folosită adesea pentru reducerea complexității proiectării unităților de control. Este posibil ca la un nivel să poată fi sincronizat masivul sub controlul unui singur flux de instrucțiuni, iar la nivelele inferioare (de exemplu, microcod) să se execute diferite secvențe de instrucțiuni. Astfel, funcție de starea locală, microcontrolerul local poate acționa autonom, deși ca răspuns la o instrucțiune identică. Un astfel de sistem a a fost propus de Pease (1977) și implementat de Baba (1987).

În realitate deciziile legate de control, ilustrate de exemplele anterioare, sînt adesea decizii inginerești luate funcție de tehnologiile de implementare. Spre deosebire de distincția dintre modul de control declarativ sau cu flux de instrucțiuni, aceste decizii nu sînt filozofice ,deoarece în practică orice formă de control poate fi simulată cu alta. Oricum se va plăti o penalitate în eficiență dacă structura de control nu este adecvată algoritmului. Execuția programelor SIMD pe mașini MIMD reprezintă o risipă de hardware, penalizată prin overhead-ul de sincronizare, în timp ce cazul opus va impune ca mașina SIMD să execute un interpreter, probabil foarte ineficient, care ar putea interpreta datele procesorului ca instrucțiuni.

Oricare ar fi alegerile, deciziile se vor lua pe baza restricţiilor tehnologice şi a domeniului de aplicaţii. De exemplu dacă se implementează un sistem cu multiplicare pentru prelucrarea imaginilor binare, o alegere bună pentru procesor va fi unul serial pe bit. O structură MIMD cu procesoare pe 1 bit nu va fi implementată eficient, datorită overhead-ului complex introdus de controler şi memoria program. De aceea o maşină SIMD va corespunde mai bine. Din păcate, pentru sistemele cu multiplicare nu există o soluţie universală, sau cel puţin nu a fost încă descoperită.

3.2.3. Distribuirea puterii de calcul

Unul din subiectele cele mai importante şi controversate privind multiplicarea este distribuirea puterii de calcul. S-au realizat proiecte la care numărul procesoarelor variază de la cîteva zeci la cîteva mii. Evident, pentru un anumit cost, cu cît este procesorul mai complex, cu atît pot fi combinate mai puţine într-un masiv. Aceasta este o problemă clasică de compromis ,între puterea procesorului individual şi dimensiunea masivului de procesoare. Care sînt atunci factorii care influenţează această decizie?

Un factor important, deja discutat în capitolul 1, este aplicabilitatea unui calculator paralel la o mulţime dată de probleme. Cu cît este calculatorul mai paralel, cu atît devine mai specializat. Aceasta este într-adevăr un argument de eficienţă deoarece dacă problema este mai puţin paralelă decît calculatorul, eficienţa acestuia va suferi. Acest argument poate fi considerat pentru o gamă largă de aplicaţii, pentru care există algoritmi paraleli; vezi cap. 5. Evident, cuantificarea paralelismului este imposibilă, cu excepţia situaţiei cînd tipul şi dimensiunea problemei sînt cunoscute. Oricum, pentru o clasă largă de probleme, cantitatea de paralelism care poate fi exploatată este mult mai mare decît cele mai mari masive constituite pînă în anul 1988. Mai mult, este probabil că dimensiunea acestor probleme va creşte cu puterea de calcul disponibilă şi, pentru un procesor de complexitate fixă, va varia cu numărul procesoarelor multiplicate

Pentru o anumită clasă de probleme, poate fi necesară o mică cantitate de paralelism, satisfăcută de masive de dimensiuni reduse construite cu procesoare foarte puternice, hardware relativ costisitor pentru operaţii în virgulă mobilă, sau de calculatoare vectoriale pipeline. Pentru alte probleme, paralelismul poate fi atins pe scară mare, luîndu-se în considerare alte aspecte.

Cel mai simplu procesor este de tipul serial pe bit ,din care, prin exploatarea tehnologiilor VLSI, pot fi combinate foarte ieftin mai multe mii. În acest mod, puterea de calcul este discretizată foarte fin, la nivelul unui bit-slice al datelor, care trebuie să se caracterizeze printr-un paralelism înalt. Se poate vedea că pentru un paralelism dat, această soluţie asigură o utilizare foarte eficientă a hardware-ului pentru toate formele de date : booleene, caractere, numere întregi sau în virgulă mobilă. Se mai poare arăta că, pentru un număr dat de elemente logice, această soluţie oferă puterea de calcul maximă, cel puţin pentru operaţii simple. Să considerăm ca problemă simplă adunarea unui număr de perechi de numere reprezentate pe b biţi (să spunem N, unde $N > b$), avînd b numărătoare

pe 1 bit cu timpul de calcul τ_{FA}. Fig. 3.2. prezintă tabela de adevăr pentru un sumator complet și o implementare a sa.

O soluție ar consta din înlănțuirea numărătoarelor, care ar adresa astfel toți biții unui cuvînt. Acesta este denumit uneori sumatorul paralel, prezentat în fig. 3.3. Se poate vedea că procesorul nu este paralel deloc, ci secvențial. Transportul de la primul sumator este definit după τ_{FA},

A	B	C_{in}	P	G	S	C_{out}
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	1	0	1	0
0	1	1	1	0	0	1
1	0	0	1	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	1
1	1	1	0	1	1	1

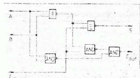


Fig. 3.2 Tabela de adevăr și schema unui sumator construit cu circuite SAU exclusiv și NAND. Intrările sînt A, B și un semnal de transport C_{in}, iar ieșirile sînt S (suma) și transportul, C_{out}. Tabela de adevăr mai prezintă și semnalele P și G, funcții numai de A și B, care indică faptul că transportul este propagat sau generat în această poziție.

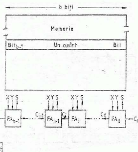


Fig. 3.3 Sumator paralel, sau cu transport propagat serial.

întîrzierea produsă de două porți NAND și una OR exclusiv. Acest transport este necesar la calculul celei de-a doua sume din lanț. Timpul total, pentru o adunare pe b biți, va fi :

$$t_r = \tau_{FA} + 2(b-1)\tau_d \tag{3.1}$$

unde τ_d este timpul de propagare al semnalului pe o poartă NAND. Ignorînd timpul de acces la memorie ,soluția se va obține înmulțind cu N ecuația (3.1). Notăm acest timp cu T_r :

$$T_r = N[\tau_{FA} + 2(b-1)\tau_d] \tag{3.2}$$

Soluția bit-slice folosește fiecare sumator ca unitate independentă care sumează în paralel perechi binare ale unor cuvinte diferite. Transportul obținut într-o etapă trebuie păstrat de un registru pentru a deveni variabilă de intrare în etapa următoare. Ilustrarea apare în fig. 3.4, iar timpul necesar adunării fiecărui bit-slice, este

$$t_b = \tau_{FA} + \tau_1 \tag{3.3}$$

unde τ_1 este intervalul de timp necesar memorării transportului. Timpul total necesar rezolvării problemei prin metoda bit-slice va fi:

$$T_b = [N/b]b(\tau_{FA} + \tau_1) \qquad (3.4)$$

unde [...] notează funcţia parte întreagă.

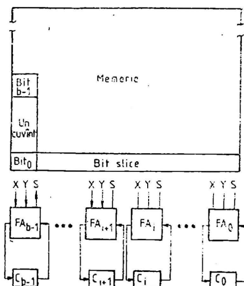


Fig. 3.4 Adunare paralelă, sau bit-slice.

Se poate vedea că timpul necesar sumării bit-slice creşte cu N/b în timp ce în cazul soluţiei anterioare varia cu N.

Este util să luăm în considerare şi timpul de acces la memorie; de exemplu, prin adunarea timpului de acces la memorie, τ_M la ambele ecuaţii (3.1) şi (3.3) se obţine următorul raport:

$$\frac{T_r}{T_b} = \frac{N[\tau_M + \tau_{FA} + 2(6-1)\tau_d]}{[N/b]b[\tau_M + \tau_{FA} + \tau_1)}$$

Dacă N este un multiplu al lui b, prin simplificare se obţine:

$$\frac{T_r}{T_b} = \frac{\tau_M + \tau_{FA} + 2(b-1)\tau_d}{\tau_M + \tau_{FA} + \tau_1}$$

Evident dacă $\tau_M \gg \tau_d$ prin soluţia bit-slice se cîştigă prea puţin. Pare foarte probabil că acesta este motivul pentru care maşinile anilor '50 au adoptat soluţia paralel pe bit, serial pe cuvînt. Este interesant de speculat cum ar fi evoluat calculatoarele dacă diferenţa dintre tehnologia memoriilor şi a circuitelor logice ar fi fost în anii '50 mai mică.

Astăzi este economic să se folosească şi pentru memorii şi pentru celelalte circuite aceeaşi tehnologie, realizîndu-le chiar integrat pe aceeaşi capsulă VLSI. Astfel, raportul definit de ecuaţia (3.5) este semnificativ mai mare decît 1, chiar pentru dimensiuni modeste ale cuvîntului. Discuţia nu se încheie aici, deoarece procesele secvenţiale pot fi accelerate adesea cu noi tehnici. Acesta este cazul recurenţelor, aşa cum se va vedea în cap. 5, unde se descriu algoritmi paraleli. Adunarea a două numere de b biţi este un caz clasic de algoritm recursiv, ce poate fi calculat în $\log_2 b$ paşi. S-a arătat că aceasta este limita inferioară (Winograd 1967).

Cum se poate realiza acest lucru cu circuite logice? Cheia o oferă definirea celor 2 stări ale unui sumator, denumite propagare şi generare, notate pe fig. 3.2 cu P, respectiv G. Funcţie de cele 2 valori de intrare care se sumează, ele definesc cele două stări: „se generează transport" şi „transportul va fi propagat". Cu o pereche de astfel de stări, se pot defini

în mod simplu noile stări de generare şi propagare. De exemplu, fiind date P_0, P_1 şi G_0, G_1, se definesc 2 stări noi P' şi G' cu :

$$G' = G_1 \vee (P_1 \wedge G_0)$$

şi (3.6)

$$P' = (P_1 \wedge P_0)$$

Cu acestea, se poate defini un transport la ieşire pe baza celui de la intrare :

$$C' = G' \vee (C \wedge P') \quad (3.7)$$

Fig. 3.5 prezintă modul de implementare a unităţii cu transport anticipat, pe baza ecuaţiilor (3.6) şi (3.7). Această unitate poate fi încorporată într-o structură de tip arbore, care formează sumatorul cu transport anticipat. Soluţia pentru cuvinte pe 8 biţi este reprezentată în fig. 3.6. Se poate vedea că, în general, sînt necesare b sumatoare şi b-1 unităţi pentru calcularea transportului. Timpul de rezoluţie pentru b >2 este determinat de timpul necesar pentru obţinerea bitului sumă cel mai semnificativ :

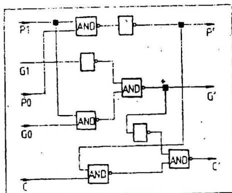


Fig. 3.5 Un circuit cu transport anticipat, construit cu porţi NAND şi inversoare, care are la intrare semnalele de propagare şi generare de la două poziţii binare (PO, GO şi P1, S1); pe baza semnalului de transport C, la ieşire se produc semnale de propagare şi generare modificate (folosite pentru conectarea în cascadă), ca şi un semnal de transport, C.

$$t_C = 2\tau_{FA} + (\log_2 b - 1)\tau_{CL} \quad (3.8)$$

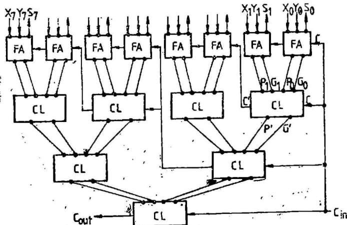


Fig. 3.6 Un sumator cu transport anticipat pe 8 biţi. CL sînt unităţile pentru producerea transportului ; FA sînt sumatoarele (vezi şi fig. 3.2 şi 3.5).

unde τ_{CL} este timpul necesar obţinerii lui C' într-o singură unitate de calcul anticipat al transportului (fig. 3.5). Pentru valorile obişnuite ale lui b, se obţine o îmbunătăţire considerabilă a timpului de execuţie, faţă de ecuaţia

(3.1), la costul dublării numărului de porţi. Pentru problema adunării a N perechi de numere pe b biţi, şi această soluţie este mai lentă decît soluţia bit-slice, aşa cum se vede din ecuaţia (3.9):

$$\frac{T_c}{T_b} = \frac{\tau_M + (\log_2 b - 1)\tau_{FA}}{\tau_M + \tau_{FA} + \tau_l} \tag{3.9}$$

În acest caz, timpul de rezoluţie al unităţilor de calcul anticipat al transportului (τ_{CL}) s-a egalat cu timpul de rezoluţie al sumatorului, τ_{FA}.

Se pot aplica tehnici similare şi pentru calculul rapid al înmulţirii (Waser 1978) dar, din nou, este necesar un număr sporit de porţi logice. Astfel, am prezentat tehnici pentru construirea unor unităţi funcţionale mai rapide, dar şi mai complexe, deşi creşterile relative ale costului şi eficienţei nu vor putea atinge niciodată valorile corespunzătoare soluţiei bit-slice (cel puţin atît timp cît nu se poate realiza $\tau \gg \tau_{FA}$).

Nu înseamnă că tot ansamblul hardware trebuie să se bazeze pe metoda bit-slice, deoarece pentru operaţiile cu un scalar trebuie folosit hardware mai complex. În acest mod, distribuţia puterii de calcul va depinde de anumiţi factori, între care cei mai importanţi sînt raportul cost/performanţă şi paralelismul anticipat al încărcării.

3.3. Reţele de comutare

3.3.1. Introducere

Teoria şi construcţia reţelelor de comutare sînt fundamentale pentru succesul paralelismului pe scară largă, fezabil acum prin exploatarea tehnologiei VLSI. Multiplicarea pe scară largă, aşa cum a fost descrisă în §2.3.1., nu este viabilă dacă nu se pot stabili conexiuni fie între procesoare, fie între procesoare şi memorie, prin program. Astfel de conexiuni pot fi stabilite cu reţele de comutare-o sumă de comutatoare, într-o configuraţie de conectare dată. Teoria reţelelor de comutaţie a fost dezvoltată la început pentru industria telefoanelor (Clos 1953, Benes 1965), dar convergenţa ei cu industria calculatoarelor, reţelelor de calculatoare, telefonia numerică şi acum cu domeniul calculatoarelor paralele au determinat creşterea interesului pentru ea. În continuare vom explora arhitectura reţelelor de comutare, cu un accent particular pe prelucrarea paralelă pe scară largă. Informaţii suplimentare pot fi găsite în lucrarea recentă a lui Siegel (1985).

Reţelele de comutare asigură un set de interconexiuni sau corespondenţe între două mulţimi de noduri, de intrare şi de ieşire. Pentru N intrări şi M ieşiri există N^M corespondenţe bine definite de la intrare la ieşire, unde prin definit bine se înţelege că fiecare ieşire este definită în termenii unei singure intrări. Fig. 3.7 ilustrează acest lucru prin toate componentele bine definite între 3 intrări şi 2 ieşiri. O reţea care realizează toate cele N^M

corespondenţe se numeşte reţea de conectare generalizată (generalised connection network-GCN).

Dacă limităm numărul conexiunilor numai la cel al corespondenţelor bijective, atunci vor fi bine definite N!. Fig. 3.7 mai prezintă sub-mulţimea corespondenţelor bijective; ele transmit datele de la o intrare numai la o ieşire. Pentru ca această clasă de corespondenţe să fie nevidă, trebuie să existe cel puţin tot atîtea intrări cît şi ieşiri. O reţea care realizează aceste N! conexiuni se numeşte reţea de conectare (connection network CN). Posibilitatea evidentă de implementare a unei reţele de conectare generalizate este cu o reţea în cruce completă,la care fiecare intrare poate fi conectată la fiecare ieşire. Fig. 3.8 ilustrează această variantă, folosind două reprezentări diferite. În prima (fig. 3.8 (a)) arcele reprezintă mulţimile de intrare şi ieşire, iar nodurile reprezintă comutatoare. O reprezentare alternativă o oferă graful bipartit (fig. 3.8 (b)), unde nodurile reprezintă mulţimile de intrare şi ieşire, iar arcele, conexiunile. Reţeaua în cruce completă este reţeaua de comutare cea mai generală, dar sînt necesare $N \times M$ comutatoare. De aceea, pentru N mare, reţeaua va deveni foarte costisitoare. În general, o limită practică pentru N este în jur de 2^7, în timp ce numărul procesoarelor ce sînt comutate poate fi de 2^{14} sau mai mult. Reţele de acest tip s-au folosit la Burroughs BSP, una cu $N=16$ şi $M=17$, iar cealaltă cu $M=16$ şi $N=17$.

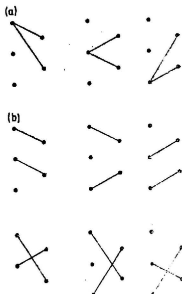


Fig. 3.7 Toate conexiunile posibile de la trei intrări la două ieşiri: (a) de la o intrare la mai multe ieşiri; b) de la o intrare la o ieşire.

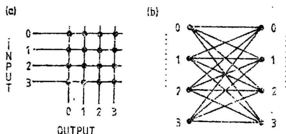


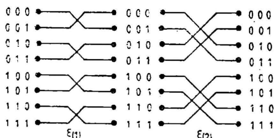
Fig. 3.8 Două reprezentări ale reţelei de comutare în cruce de la 4 intrări la 4 ieşiri.

Alte reţele de comutare se bazează fie pe reţele în cruce unice fie pe reţele în cruce incomplete. Acest aspect va fi discutat ulterior, dar va trebui să introducem întîi o notaţie şi să definim nişte permutări care vor simplifica considerabil discuţia.

3.3.2. Cîteva permutări fundamentale

O permutare pe o mulţime ordonată de N noduri se poate defini cu o funcţie $\pi(x)$, unde x şi $\pi(x)$ trebuie să fie de asemenea bijectivă. De exemplu, funcţia:

$$\varepsilon_{(k)}(x) = \left\lfloor \frac{x}{2^k} \right\rfloor 2^k + \left| |x|_{2^k} + 2^{k-1} \right|_{2^k} \tag{3.10}$$



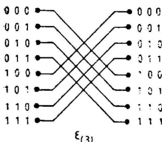


Fig. 3.9 Permutări cu schimbare.

defineşte o permutare cu schimbare (fig. 3.9)

Totuşi, pentru multe permutări se găseşte adesea că se poate obţine un mod mai simplu de definire a permutării pe baza reprezentării binare a lui x. Astfel

$$x = (b_n, b_{n-1}, \ldots, b_1) =$$
$$= b_n 2^{n-1} + b_{n-1} 2^{n-2} + \ldots$$
$$\ldots + b_1 \quad 0 \leqslant b_i \leqslant 1$$

reprezintă adresa binară a unui element din mulţime. Se pot defini permutări cu adresele lor binare. Această notaţie a fost folosită cu un efect salutar în mai multe lucrări referitoare la manipularea datelor şi reţelelor de transfer (Flanders 1982, Parker 1980, Nassimi Sahni 1980).

(i) *Permutarea cu schimbare.*

Permutarea cu schimbare definită mai sus, poate fi definită acum mai simplu în termenii reprezentării binare a lui x

$$\varepsilon_{(k)}(x) = \{b_n, \ldots, \bar{b}_k, \ldots, b_1\} \tag{3.11}$$

Bara notează complementul bitului corespunzător. Astfel se poate defini a K-a permutare cu schimbare prin complementarea bitului de rang K din reprezentarea lui x.

(ii) *Permutarea cu amestec perfect* (*perfect shuffle*)

Această permutare este astfel denumită deoarece poate fi realizată prin împărţirea mulţimii în două şi intercalarea celor două mulţimi obţinute ca în cazul amestecului perfect de cărţi de joc (vezi fig. 3.10). Această permutare corespunde la deplasarea circulară la stînga a unei unităţi în reprezentare binară a lui x.

$$\sigma(x) = \{b_{n-1}, b_{n-2}, \ldots, b_1, b_n\} \tag{3.12}$$

Se pot defini de asemenea, pe baza deplasării la stînga ciclice cu K biţi, cei mai puţin semnificativi sau cei mai semnificativi, subamestecul K, $\sigma_{(k)}$, (subshuffle) şi superamestecul $\sigma^{(k)}$ (supershuffle).

$$\sigma_{(k)}(x) = \{b_n, \ldots, b_{k-1}, b_{k+1}, \ldots, b_1, b_k\} \tag{3.13}$$

$$\sigma^{(k)}(x) = \{b_{n-1}, \ldots, b_{n-k+1}, b_n, b_{n-k}, \ldots, b_1\} \tag{3.14}$$

Şi acestea sînt prezente în fig. 3.14 pentru n=3 şi k=2. În mod limpede

$$\sigma^{(n)}(x) = \sigma_{(n)}(x) = \sigma(x)$$

şi

$$\sigma^{(1)}(x) = \sigma_{(1)}(x) = x.$$

Se mai observă în fig. 3.10 că subamestecurile (biţii cei mai puţin semnificativi) tratează mulţimea ca un număr de submulţimi, executînd un amestec perfect cu fiecare. Superamestecurile, pe de altă parte, modifică o mulţime întreagă, dar cresc numărul de conexiuni.

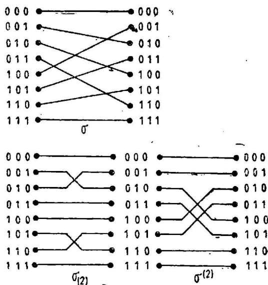


Fig. 3.10 Permutări cu amestec perfect.

(iii) *Permutarea fluture*

Permutarea fluture (fig. 3.11) este definită peste reprezentarea binară a lui x cu schimbarea primilor şi ultimilor biţi

$$\beta(x) = \{b_1, b_{n-1}, \ldots b_2, b_n\} \tag{3.15}$$

Ca şi la permutările cu amestec se pot defini al K-lea subfluture sau al K-lea super-fluture a lui x. În primul caz e schimbarea primilor biţi cu cei de pe poziţia K şi mai departe, în timp ce pentru al doilea caz se schimbă biţii n cu (n-k-1).

$$\beta_{(k)}(x) = b_n, \ldots, b_{k+1}, b_1, b_{k-1}, \ldots, b_k\} \tag{3.16}$$

$$\beta^{(k)}(x) = \{b_{n-k+1}, \ldots, b_{n-k+2}, b_n, b_{n-k}, \ldots, b_1\} \tag{3.17}$$

Şi aceste permutări se prezintă în fig. 3.11, pentru n=3 şi k=2. Din nou

$$\beta^{(n)}(x) = \beta_{(n)}(x) = \beta(x)$$

şi

$$\beta^{(1)}(x) = \beta_{(1)}(x) = x.$$

(IV) *Permutarea cu inversarea biţilor*

Această permutare, după cum sugerează şi denumirea, se defineşte peste reprezentarea binară a lui x prin inversarea ordinii biţilor:

$$\rho(x) = \{b_1, b_2, \ldots, b_n\} \qquad (3.18)$$

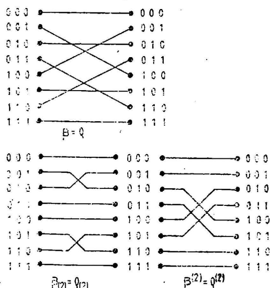


Fig. 3.11 Permutări fluture şi cu inversarea biţilor.

O aplicaţie unde intervine această permutare este algoritmul transformatei F rapidă (vezi § 5.5).

Ca şi pentru cele două permutări anterioare se pot defini două variante ale acestei permutări care afectează cei mai puţin semnificativi K biţi sau cei mai semnificativi K biţi („sub bit reversal" şi respectiv „super bit reversal").

$$\rho_{(k)}x = \{b_n, \ldots, b_{k+1}, b_1, b_2, \ldots, b_k\} \qquad (3.19)$$

$$\rho^{(k)}(x) = \{b_{n-k+1}, b_{n-k+2}, \ldots, b_n, b_{n-k}, \ldots, b_1\} \qquad (3.2$$

Fig. 3.11 prezintă şi acest tip de permutare pentru n=3. Totuşi, cele 2 permutări nu sînt întotdeauna echivalente, aşa cum se va arăta mai tîrziu, în cadrul subcapitolului de algebră a permutărilor.

(V) *Permutarea cu deplasare*

Această ultimă permutare se descrie mult mai uşor fără a folosi reprezentarea binară a lui x:

$$\alpha(x) = |x + 1|_{2^n} \qquad (3.21)$$

Definim din nou sub-deplasarea cu biţii cei mai puţin semnificativi şi super-deplasarea cu biţii cei mai semnificativi. În termenii reprezentării binare a lui x, ecuaţia (3.21) defineşte adunarea binară peste cîmpul cu n biţi, ignorînd depăşirea. Prin urmare, sub-deplasarea şi super-deplasarea pot fi definite astfel:

$$\alpha_{(k)}(x) = |x + 1|_{2^k} + \left[\frac{x}{2^k}\right]_{2^k}. \qquad (3.22)$$

$$\alpha_{(k)}(x) = |x + 2^{n-k}|_{2^n}. \qquad (3.23)$$

Acestea sînt prezentate în fig. 3.12 pentru n=3 şi k=2.

3.3.3. Algebra permutațiilor

Pentru a defini permutațiile mai complexe și echivalente vom dori să combinăm și să manipulăm permutări simple descrise mai sus. Algebra funcțiilor descrie tocmai aceasta. Combinarea a două funcții se notează cu $\rho(\sigma(x))$, ceea ce definește o permutare în termenii unui amestec perfect urmat de o inversare a biților. Folosind grafuri bipartite, aceasta e echivalentă cu conectarea ieșirilor amestecului la inversarea biților. Pentru a simplifica expresia și în același timp pentru a păstra sensul deplasării informațiilor de la stînga la dreapta în grafurile bipartite, se va abrevia expresia de mai sus cu $\sigma\rho$. Precedentul pentru aceste notații este combinația operatorilor în APL (Iverson 1979). De asemenea, $\sigma\sigma$ va fi abreviat cu σ^2.

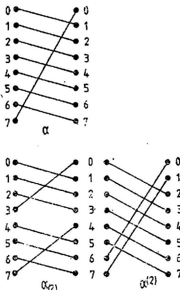


Fig. 3.12 Permutări cu deplasare.

Ca în orice algebră trebuie definită o unitate sau permutare identică pe care o notăm cu i. Această permutare păstrează ordinea mulțimii de intrare:

$$i(x) = x \tag{3.24}$$

Avînd definiția permutării identică, se poate defini inversa unei aplicații (mapping). De exemplu, inversa permutării cu amestec perfect se notează cu σ^{-1} și $\sigma \cdot \sigma^{-1} = \sigma^{-1} \cdot \sigma = i$. Aplicația inversă poate fi înțeleasă prin urmărirea grafului bipartit de la dreapta la stînga.

Folosind algoritmul permutării vom stabili cîteva identități importante. În cele mai multe cazuri aceste identități pot fi verificate folosind reprezentarea binară a lui x.

Prima identitate se referă la relația dintre sub- și super- permutare, unde pentru permutarea generală π,

$$\pi^{(k)} = \sigma^k \pi_{(k)} \sigma^{-k} \tag{3.25}$$

Prezentăm în continuare un grup de identități care se referă la inversa permutării definite anterior:

$$\varepsilon_{(k)}^{-1} = \varepsilon_{(k)} \tag{3.26a}$$

$$\rho_{(k)}^{-1} = \rho_{(k)} \tag{3.26b}$$

$$\beta_{(k)}^{-1} = \beta_{(k)} \tag{3.26c}$$

$$\alpha_{(k)}^{-1} = \alpha_{(k)}^{k-1} \tag{3.26d}$$

$$\sigma_{(k)}^{-1} = \sigma_{(k)}^{k-1} \tag{3.26e}$$

Un alt grup de identităţi se referă la relaţiile ce se pot defini între diferite permutări:

$$\varepsilon_{(k)} = \sigma_{(k)}\alpha_{(1)}\sigma_{(k)}^{-1} \quad (3.27a)$$

$$\sigma_{(k)} = \beta_{(1)}\ldots\beta_{(k)} \quad (3.27b)$$

$$\rho_{(k)} = \sigma_{(1)}\ldots\sigma_{(k)} \quad (3.27c)$$

Ultimul grup de identităţi caracterizează numai prima sub-permutare:

$$\sigma_{(1)} = i \quad (3.28a)$$

$$\rho_{(1)} = i \quad (3.28b)$$

$$\beta_{(1)} = i \quad (3.28c)$$

Ca un exemplu al utilizării algebrei şi identităţilor definite să stabilim ce a fost evident în fig. 3.11 pt. $n=3$, $\rho = \beta$. Din ecuaţia (3.27 b) şi (3.27c)

$$\rho_{(3)} = \beta_{(1)}\beta_{(1)}\beta_{(2)}\beta_{(1)}\beta_{(2)}\beta_{(3)}$$

Deoarece $\beta_{(1)}$ este permutare identică (ec. 3.28c) şi $\beta_{(k)}$ este propriul invers (ec. 3.26 c)

$$\rho_{(3)} = \beta_{(3)}$$

3.3.4. Reţele cu un singur etaj

Reţelele cu un singur etaj constau dintr-un etaj fix sau unic de comutatoare. Astfel, pentru un masiv cu P procesoare, reţeaua ar consta dintr-un masiv unic de selectoare cu P căi. Acest tip de reţea este distribuită cu uşurinţă, selectorii fiind elemente hardware ale procesoarelor. De obicei acest tip de reţea este asociat comutatorului inter-procesoare din fig. 3.1.

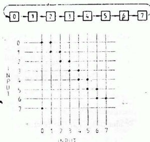


Fig. 3.13 Reţeaua inel $R_{(1)}$.

În mod normal, reţelele cu un singur etaj stabilesc direct un număr limitat de permutări. Alte permutări mai generale, pot fi realizate iterativ. Astfel, de exemplu, dacă reţeaua asigură permutarea cu deplasare α, pentru a realiza $\alpha^{(n-k)}$ unde $\alpha^{n-k} = \alpha^{2^k}$ sînt necesare 2^k iteraţii prin reţea.

Vom descrie în continuare un număr de reţele cu un singur etaj, definind comutatoarele în termenii mulţimii de permutări de care le pot genera. Astfel:

$$SW = \{\pi_1, \pi_2, \ldots, \pi_m\}$$

defineşte comutatorul SW, care execută m permutări, π_1 sau π_2 pînă la π_m.

În discuţia ce urmează vom presupune ca $P = 2^p$, dacă nu se specifică altceva.

(i) *Rețeaua cu inel*

Aceasta este cea mai simplă rețea; este formată dintr-un inel de procesoare, cu un singur sens de circulație a informațiilor. Fig. 3.17 ilustrează această rețea și se poate observa că avem de-a face cu o rețea cu accesiuni încrucișate incompletă definită ca:

$$R = \{i, \alpha_{(p)}\}$$

(ii) *Rețeaua vecinătății proxime*

Aceasta este o extensie simplă a rețelei în inel care permite o circulație bi-direcțională a informațiilor. Se ilustrează în fig. 3.14 și se definește astfel:

$$NN = \{\alpha_{(p)}^{-1}, i, \alpha_{(p)}\}$$

Această rețea este simplă și ieftină dar nu este indicată pentru un număr mare de procesoare. Oricum, poate fi generalizată la mai mult de o dimensiune. De exemplu, dacă $P = Q^k$ și $Q = 2^q$, atunci rețeaua K-dimensională poate fi definită astfel:

$$NN_{(k)} = \{\sigma_{(lq)}^{q}\alpha_{(q)}^{-1}\sigma_{(lq)}^{q}, i, \sigma_{(lq)}^{q}\alpha_{(q)}\sigma_{(lq)}^{q} : l = 1, \ldots k\}$$

Calculatorul ICL DAP folosește acest tip de rețea cu $k = 2$ și $q=6$.

$$S_{DAP} = \{\sigma_{(12)}^{6}\sigma_{(6)}^{-1}, \alpha_{(12)}^{-6}\alpha_{(6)}^{-1}, i, \alpha_{(6)}\ \sigma_{(12)}^{6}\alpha_{(6)}\sigma_{(12)}^{-6}\}$$

Aici permutările pot fi considerate ca deplasări pe direcția nord, sud, est și vest pe o grilă bidimensională, conectată la capete. Și ILLIAC IV are o astfel de rețea bidimensională cu $q = 3$, deși aici periodicitatea se definește cumva diferit:

$$S_{14} = \{\alpha_{(6)}^{3}\alpha_{(3)}^{-1}\sigma_{(6)}^{-3}, \alpha_{(6)}^{-1} i, \alpha_{(6)}, \sigma_{(6)}^{3}, \alpha_{(3)}\sigma_{(6)}^{-6}\}$$

Este interesant de observat ce se întîmplă cînd considerăm cazul limitativ al rețelei cu $k = p = \log_2 l$. Aici $q = 1$ și din (3.2.7a)

$$NN_{(p)} = \{\varepsilon_{(l)}, i : l = 1, \ldots, k\}$$

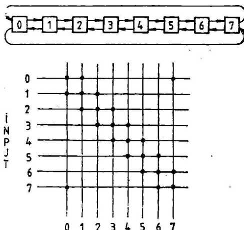


Fig. 3.14 Rețeaua vecinătății proxime $NN_{(l)}$

Astfel este descris hipercubul binar a cărui structură constă dintr-un hipercub cu p dimensiuni și 2 procesoare pe fiecare muchie. Fig. 3.15 exemplifică structura cu $p = 3$, unde se observă că schimbarea k este echivalentă schimbării inferioare peste planul de schimbare al k-lea (e prezentat planul $k = 1$)

(iii) *Rețelele cu amestec perfect*

Amestecul perfect reprezintă permutarea puternică pe care se poate baza o rețea (Stone 1971, Lang și Stone 1976, Lang 1976). Definim mai jos comutatorul cu amestec perfect, așa cum este prezentat în fig. 3.26 :

$$PS = (\sigma^{-1}_{(p)}, i, \sigma_{(p)})$$

Se poate observa că acesta nu este un comutator foarte mulțumitor, deoarece lasă neconectate 4 submulțimi de procesoare. De aceea, s-au propus rețelele cu schimbare și amestec perfect (Stone 1971) și cu vecinătate proximă și amestec perfect (Grosch 1979). Acestea sînt definite mai jos și ilustrate în fig. 3.17 pentru $N = 8$

$$PSE = \{\sigma^{-1}_{(p)}, i, \varepsilon_{(1)}, \sigma_{(p)}\}$$

$$PSSN = \{\sigma^{-1}_{(p)}, \alpha^{-1}_{(p)}, i, \alpha_{(p)}, \sigma_{(p)}\}$$

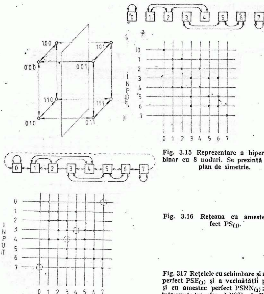


Fig. 3.15 Reprezentare a hipercubului binar cu 8 noduri. Se prezintă primul plan de simetrie.

Fig. 3.16 Rețeaua cu amestec perfect $PS_{(1)}$.

Fig. 317 Rețelele cu schimbare și amestec perfect $PSE_{(1)}$ și a vecinătății proxime și cu amestec perfect $PSNN_{(1)}$; liniile întrerupte transformă $PSE_{(1)}$ în $PSNN_{(1)}$.

Ambele rețele pot fi generalizate pentru mai mult de o dimensiune, la fel ca pentru rețeaua vecinătății proxime. Totuși în cazul limită, k = p, ambele sînt echivalente cu hipercubul binar.

3.3.5. Cîteva proprietăți ale rețelelor cu un singur etaj.

Rețele sau comutatoarele cu un singur etaj pot fi folosite iterativ pentru a simula o rețea de conectare care așa cum s-a menționat anterior, poate realiza toate cele P! aplicații bijective ale intrărilor la ieșiri. Prin iterație înțelegem că după un pas prin rețea, ieșirea la un procesor dat va deveni intrarea aceleiași sau altei structuri a rețelei. Astfel pentru o rețea dată este important să știm cît de multe iterații vor fi necesare pentru a defini o structură de comunicație. Viteza de calcul este influențată, mai ales dacă este necesar un număr mare de iterații.

Pentru a compara rețelele discutate anterior definim conceptul de distanță în rețea. De exemplu, distanța între nodurile i și j în rețea se definește ca numărul de iterații necesare pentru a stabili aplicația $i \to j$. Apoi definim parametrii distanței, care sînt o funcție a rețelei date și dau informații despre eficiența rețelei. Acești parametrii au fost inițial definiți de Jesshope (1980b, 1980c) și au fost folosiți pentru estimarea timpilor necesari transferului informațiilor în masivele de procesoare (Jesshope 1980a, 1980c). Un studiu mai complet privind simularea unei rețele de către alta se poate afla în lucrarea lui Siegel (1985).

(i) *Măsura distanței maxime*

Măsura distanței maxime $D_{(k)}$ pentru o rețea k-dimensională, dă distanța maximă între oricare 2 noduri din rețea. Reprezintă o măsură a eficienței rețelei pentru probleme care implică comunicație globală. Într-o dimensiune a unei rețele R, procesorul care este cel mai depărtat de procesorul i este $i-1$, iar pentru rețeaua NN este procesorul $i+N/2$. Jesshope (1981b) dă un argument inductiv care introduce reducerea la scară prin K, folosită în tabelul 3.1. Pentru rețeaua cu amestec perfect este mai ușor de folosit reprezentarea binară a adresei nodurilor. Astfel pentru PSE, nodul $i=(b_q, b_{q-1}, \ldots, b_1)$ este cel mai depărtat de nodul $j = \{\bar{b}_q, \bar{b}_{q-1}, \ldots, \bar{b}_1\}$. Această aplicație poate fi realizată în q-1 amestecuri perfecte (deplasarea biților la stînga) și q schimbări de ordinul 1 (complementarea primului bit). Pentru rețeaua PSNN, nodurile cele mai îndepărtate sînt i și $i+j$, unde $j = (10101\ldots)$. Din nou, sînt necesare q-1 amestecuri perfecte dar numai q/2 deplasări (adună 1 la i). Evident, distanța maximă în rețeaua PS este infinită. Toate aceste rezultate sînt prezentate în tabelul 3.1.

Tabelul 3.1. **Proprietăți ale rețelelor cu un singur etaj.**

Proprietatea	Rețea				
	R_k	$NN_{(k)}$	$PS_{(k)}$	$PSE_{(k)}$	$PSNN_{(k)}$
Distanța maximă $D_{(k)}$	$k(Q-1)$	$kQ/2$	∞	$k(2q-1)$	$k(q+q/2-1)$
Fan out $F_{(k)}$	$k(Q-1)$	$k(Q-1)$	∞	$k(2q-1)$	$k(2q-1)$

(ii) *Parametrul fan out, fan in*

Parametrul fan out, fan in reprezintă o evaluare a distanţei necesare pentru propagarea unei informaţii la toate nodurile din reţea sau, invers, reducerea (colectarea) informaţiei. Acest parametru este important pentru mulţi algoritmi „divide et impera" (vezi capitolul 5). Vor fi necesare un număr de permutări paralele (posibil sub controlul boolean). De exemplu, pentru o dimensiune a reţelelor NN sau R, un astfel de algoritm este prezentat în fig. 3.18. Se poate observa că pentru o dimensiune, numărul total de permutări cu deplasare este $Q-1=1+2+4\ldots+Q/2$. Pentru reţelele PSE şi PSNN, este indicat să se folosească reprezentarea binară a adresei sau identificatorului fiecărui procesor. Algoritmul este reprezentat schematic în fig. 3.19, unde se arată propagarea datelor de la procesorul numărul 0 la toate celelalte. Să considerăm întîi reţeaua PSE: permutările cu amestec şi schimbare sînt definite în § 3.3.2 cu ecuaţiile (3.11) şi (3.12). Aceste ecuaţii complementează bitul cel mai puţin semnificativ al adresei unui procesor şi execută o deplasare circulară la stînga a adresei respective. Problema poate fi specificată mai uşor ca una de generare a adreselor tuturor procesoarelor plecînd de la adresa sursă prin complementarea bitului cel mai puţin semnificativ al adresei şi deplasări circulare la stînga.

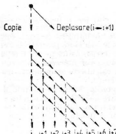


Fig. 3.18 Ilustrare a algoritmului de transfer al unei date la toate nodurile unui inel $R_{(1)}$ sau ale unei reţele cu vecinătate proximă $NN_{(1)}$.

Se poate deduce intuitiv o limită inferioară pentru numărul de paşi necesari finalizării acestei operaţii. Cu operaţiile care sînt disponibile, adresa „cea mai îndepărtată" de adresa sursă este cea obţinută prin complementarea ei. Complementarea unei adrese cu n biţi necesită n operaţii de complementare (numai a bitului cel mai puţin semnificativ) şi n-1 operaţii de deplasare, rezultînd un total de 2n-1 operaţii. Fig. 3.19 arată că prin mascări corespunzătoare se pot genera toate adresele prin complementarea adresei sursă.

Implementarea acestui algoritm pe reţeaua PSE necesită numai jumătatea numărului de operaţii de schimbare, fie prin deplasarea la stînga sau la dreapta, funcţie de paritatea sursei datelor. Prin urmare, se poate folosi acelaşi algoritm şi pentru reţeaua PSNN. Schimbările marcate pare sau impare pot fi simulate în acelaşi timp folosind deplasările la stînga sau la dreapta ale reţelei NN. Într-o formă diferită, algoritmul apare în fig. 3.20, iar rezultatele anterioare sînt ordonate în tabelul 3.1.

Pentru a compara eficienţa reţelelor descrise, vom analiza toate configuraţiile posibile ale unui masiv cu 4096 procesoare. Vom avea nevoie şi de o estimare a costului unei anumite configuraţii. Vom considera o funcţie de cost C, definită prin numărul permutărilor ce pot fi realizate de reţea, excluzînd permutarea identitate (de exemplu, C=m−1). În tab. 3.2. apar evaluările lui $D_{(k)}$ şi C pentru exemplul cu 4096 procesoare.

Se poate vedea din tab3.2.că rețelele PSE și PSNN sînt cele mai eficiente pentru k=1, dar că rețelele R și NN devin mai avantajoase pentru k=2. În continuare creșterea dimensiunii influențează negativ comportarea rețelelor R și NN. Rețelele PSE și PSNN nu produc beneficii la creșterea

Schimb Copie Amestec

Fig. 3.19 Ilustrarea algoritmului de transfer al unei date la toate nodurile unei rețele cu schimbare și amestec perfect $PSE_{(1)}$.

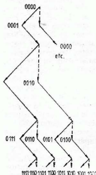


Fig. 3.20 Diagramă ce prezintă operațiile de transfer în cazul unei rețele cu schimbare și amestec. În partea superioară a tabelului se prezintă adresele binare și zecimale ale procesoarelor, iar operațiile executate, în dreapta tabelului. Cruciulițele reprezintă propagarea informației de la adresa 0.

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	
1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	Binary
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Decimal
															X	Exchange
														X	X	Shuffle
													X	X	X	Exchange
												X	X	X	X	Shuffle
									X		X	X	X	X	X	Exchange
								X	X	X	X	X	X	X	X	Shuffle
	X		X		X		X	X	X	X	X	X	X	X	X	Exchange
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Completed

Tabelul 3.2. **Distanța maximă D la un masiv cu 4096 procesoare, Funcția de cost C este trecută în paranteze.**

k	Q	$R_{(k)}$	$NN_{(k)}$	$PSE_{(k)}$	$PSNN_{(k)}$
1	4096	4095(1)	2048(2)	23(3)	17(4)
2	64	126(2)	64(4)	22(6)	16(8)
3	16	45(3)	24(6)	21(9)	15(12)
4	8	28(4)	16(8)	20(12)	16(16)
6	4	18(5)	12(12)	18(18)	12(24)
12	2	12(6)	12(12)	12(12)	12(12)

dimensiunii, pînă cînd k=p, cînd se poate observa că nici o rețea nu este mai bună ca celelalte. Într-adevăr se poate arăta cu ușurință, folosind algebra permutărilor din § 3.3.3. că cele patru rețele sînt echivalente; diferențe de cost reflectă lărgimea de bandă crescută datorită permutărilor echivalente multiple.

Probabil nu este surprinzător că rețeaua cea mai des întîlnită în cadrul masivelor de procesoare este rețeaua NN bi-dimensională, ca la ILLIAC IV (McIntyre 1970), ICL DAP (Flanders et al 1977), Goodyear MPP (Batcher 1980), GEC GRID (Robinson și Moore 1982), NTT AAP (Komdo et al 1983), LIPP de la Universitatea Linkoping (Ericsson și Danielson 1983), NCR GAPP (NCR 1984) și RPA de la Universitatea Southampton (Jesshope et al 1986). O excepție de dată recentă o reprezintă connection machine (Hillis 1985) care folosește cel puțin la nivelul VLSI, cazul limită al oricăreia din aceste rețele, cu k=p. Această rețea este denumită adeseori hipercubul binar sau doar rețeaua cub (vezi fig. 3.15).

Ne putem întreba de ce rețeaua $PSE_{(1)}$ nu este preferată, deoarece reprezintă o soluție foarte eficientă pentru comunicațiile la distanță mare. Probabil există mai multe răspunsuri referitoare atît la utilizare, cît și la implementare. Mulți algoritmi necesită comunicații locale, de obicei în forma vecinătății proxime, iar rețeaua PSE are nevoie de $2\log_2(N-1)$ iterații pentru simularea unei rețele NN bi-dimensionale (Siegel 1985). Aceasta este, de asemenea și distanța maximă în rețea. Se poate verifica ușor acest lucru prin luarea în considerare a situației că într-o rețea NN, 0 și N sînt adiacente și totuși sînt cele mai îndepărtate în rețeaua PSE. Rețeaua PSNN depășește unele din aceste limite, de aceea rețeaua PSNN bidimensională a fost aleasă în mai multe lucrări științifice; posedă conexiunea vecinătăților proxime folosită în mulți algoritmi, dar are și proprietăți suplimentare de comunicație la distanță mare. Motivul că această rețea nu a fost folosită, după cunoștințele noastre, în nici o implementare practică, constă în aspectele ridicate de implementarea propriu-zisă.

Spre deosebire de rețelele NN, cele care se bazează pe permutarea amestecului perfect nu pot fi partiționate. Dacă împărțiți o rețea $NN_{(1)}$ în jumătate, poate fi conectată numai cu două fire; chiar și o rețea $NN_{(2)}$ împărțită necesită numai $N^{1/2}$. Ce este mai important, fiecare jumătate a rețelei NN va funcționa ca o rețea NN de dimensiune redusă. Totuși, dacă orice rețea care conține permutarea amestecului perfect este împărțită în două, vor fi necesare N/2 fire pentru conectarea fiecărei jumătăți și nici una nu va putea funcționa independent. Implicațiile sînt : în proiectare, deoarece dacă rețeaua nu poate fi partiționată, trebuie proiectată ca ansamblu, existînd puțină regularitate la nivelul submodulelor; în al doilea rînd, privind fiabilitatea, aceasta realizîndu-se prin redundanță, dar fără regularitate, ceea ce este foarte scump; în al treilea rînd, privitor la numărul firelor dintre submodule, deoarece în sistemele moderne acestea sînt mai costisitoare și influențează negativ performanța. Toate cele trei aspecte cresc în importanță în era dispozitivelor VLSI (vezi cap. 6).

3.3.6. Rețele cu mai multe etaje

În unele situații este necesar să se comute un set de resurse la altul, astfel ca orice membru al unei mulțimi să poată accesa fiecare membru al celeilalte mulțimi. Un astfel de exemplu se întîlnește la conectarea procesoarelor la blocurile de memorie, ca în fig. 3.1, unde dorim să creem un sistem multiprocesor cu acces, complet, la memoria partajată. Poate fi necesar ca un anumit procesor să aibă nevoie de acces la fiecare bloc de

memorie din sistem. Pentru aceasta este necesară o rețea de conectare completă.

Deja am întîlnit rețeaua cu accesări încrucișate (fig. 3.8), care este o rețea completă și am subliniat dezavantajul ei principal — că numărul de porți necesare pentru implementare crește cu pătratul intrărilor. Pentru a înțelege mai bine, să o considerăm ca alternativă la rețelele cu un etaj, considerate în secțiunea anterioară, pentru a conecta 4096 procesoare. Ar fi necesari cel puțin 16 milioane tranzistori, dar, probabil, de mai multe ori această valoare pentru un comutator cu o performanță rezonabilă. Comutatorul NN bi-dimensional, pe de altă parte, poate fi implementat cu aproximativ 25000 tranzistori.

De asemenea, rețelele cu mai multe etaje pot reprezenta o alternativă mai ieftină la comutatorul în cruce, atunci cînd este necesară o interconectare completă. Aceste rețele folosesc un număr de comutatoare în cruce, cele mai obișnuite construindu-se pe baza unor comutatoare 2×2. Acest comutator, reprezentat în fig. 3.21, poate genera două permutări ca și două structuri folositoare pentru modul emisie (broadcasting).

Dacă pentru controlul comutatorului se folosește un bit, atunci vor fi selectate numai permutările din fig. 3.21 (a, b). Cu un masiv de N/2

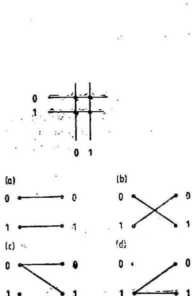


Fig. 3.21 Un element de comutare 2×2 și conexiunile posibile.

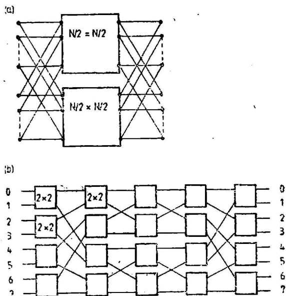


Fig. 3.22 Rețeaua Benes: (a) reducerea unui comutator în cruce N×N la două comutatoare în cruce (N/2) × (N/2) și două comutatoare cu schimbare; (b) rețeaua binară Benes cu reducere și simplificare completă (N = 8).

astfel de comutatoare putem defini un comutator cu schimbare de ordinul k, o rețea cu un singur etaj care necesită N/2 biți de control:

$$E_{(k)} = \{\varepsilon_{(k)}, i\} = \sigma_{(k)} E_{(1)} \sigma_{(k)}^{-1} \tag{3.29}$$

Dacă se folosesc doi biți de control pentru fiecare comutator 2×2, se pot genera toate cele 4 posibilități de comunicare care apar în fig. 3.21. În acest

mod putem extinde ecuația (3.29) pentru a obține comutatorul cu schimbare generalizat de ordinul K, care necesită N biți de control.

$$GE_{(k)} = \{\varepsilon_{(k)}\, 1_{(k)} u_{(k)},\, i\} = \sigma_{(k)} GE_{(1)} \sigma_{(k)}^{-1} \qquad (3.30)$$

Aici $1_{(k)}$ și $u_{(k)}$ sînt structurile inferioară și superioară pentru emisie, care se definesc cu :

$$1_{(k)}(x) = \{b_n, \ldots, b_{k+1}, 1, b_{k-1}, \ldots, b_1\}$$

$$u_{(k)}(x) = \{b_n, \ldots, b_{k+1}, 0, b_{k-1}, \ldots, b_1\}$$

(i) *Rețele conexiune*

Benes (1965) a demonstrat că o rețea cu accese încrucișate completă $N \times N$ poate fi redusă la două astfel de rețele $N/2 \times N/2$ împreună cu două rețele cu schimbare a N intrări. Fig. 3.22(a) prezintă acest rezultat, iar dacă notăm rețeaua cu accesuri încrucișate $N \times N$ cu $X_{(n)}$, unde $n = \log_2 N$, atunci aceasta poate fi descrisă cu :

$$X_{(n)} = E_{(n)} X_{(n-1)} E_{(n)} \qquad (3.31)$$

Evident, această reducere poate continua recursiv

$$X_{(n)} = E_{(n)} E_{(n-1)} E_{(n-2)} \ldots E_{(2)} X_{(1)} E_{(2)} \ldots E_{(n)} \qquad (3.32)$$

Folosind relația din ecuația (3.23) se obține :

$$X_{(n)} = \sigma_{(n)} E_{(1)} \sigma_{(n)} \ldots \sigma_{(2)}^{-1} E_{(1)} \sigma_{(2)} \ldots \sigma_{(n)} E_{(1)} \sigma_{(n)}^{-1}$$

Se poate simplifica în continuare, observînd că pre- și post- permutarea intrare și ieșire ale unei rețele nu fac decît săredefinească ordinea acestei mulțimi. Astfel se obține următoarea expresie pentru rețeaua Benes binară (Lenfant 1978) :

$$B_{(n)} = X_{(n)} = E_{(1)} \sigma_n^{-1} E_{(1)} \sigma_{(n-1)}^{-1} \ldots \sigma_{(2)}^{-1} E_{(1)} \sigma_{(2)} \ldots \sigma_{(n)} E_{(1)} \qquad (3.33)$$

Rezultatul se prezintă in figura 3.22(b). Prin definiție aceasta este o rețea conexiune totală ce asigură N ! permutări posibile ; mai mult, dacă generalizăm folosind $GE_{(1)}$, toate cele N^N aplicații bine definite pot fi executate.

(ii) *Rețele cu schimbare și amestec*

O altă clasă de rețele care nu sînt conexiune totală sînt cele denumite cu schimbare și amestec. Aici vom prezenta 4 astfel de rețele, omega (Lawrie 1975), cubul binar n-dimensional indirect (Pease 1977), banyan Goke și Lipowski 1973) și R (Parker 1980). În continuare le-am notat cu Ω, C, Y și R :

$$\Omega_{(n)} = (\sigma_{(n)} E_{(1)})^n$$

$$C_{(n)} = E_{(1)} \beta_{(2)} E_{(1)} \beta_{(3)} \ldots \beta_{(p)} E_{(1)} \sigma_p^{-1}$$

$$Y_{(n)} = E_{(1)} \beta_{(2)} E_{(1)} \beta_{(3)} \ldots \beta_{(p)} E_{(1)}$$

$$R_{(n)} = E_{(1)} \sigma_{(n)}^{-1} E_{(1)} \sigma_{(n-1)}^{-1} \ldots \sigma_{(2)}^{-1} E_{(1)} \sigma_{(n)}$$

Se observă că rețeaua banyan este cubul binar n-dimensional fără permutarea finală și că rețeaua R provine din prima jumătate a rețelei Benes binară, urmată de un amestec. O altă observație importantă demonstrată de Parker (1980) este relația dintre Ω, C și R, care verifică următoarea identitate :

$$\Omega_{(n)}^{-1} = C_{(n)} = R_{(n)} = Y_{(n)}\sigma_{(n)}$$

În fig. 3.23 se prezintă rețelele omega și cubul binar n-dimensional. Deși aceste rețele nu realizează conversiunea totală, asigură o clasă foarte bogată de permutări folosite pentru multe alte aplicații ale masivelor de procesoare. Pe lîngă operații cu matrici sînt foarte utile pentru FFT și algoritmii înrudiți (Pease 1969) ca și pentru tehnicile de sortare (Batcher 1968). Pentru mai multe detalii privind rețelele cu mai multe etaje recomandăm lucrarea lui Siegel (1985).

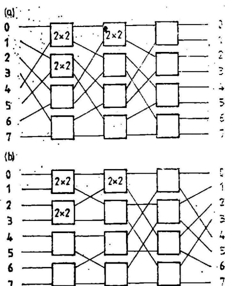


Fig. 3.23 Rețele cu schimbare și amestec : (a) rețeaua omega ; (b) rețeaua n-cub binară indirectă.

3.3.7. Controlul rețelelor

Pînă acum nu am spus nimic despre modul cum se programează comutatoarele sau selectoarele unei rețele cu un singur sau mai multe etaje. Pentru o rețea cu un singur etaj care poate realiza m permutări cu P procesoare, sînt necesari $P\log_2 m$ biți de control pentru controlul comutatorului. De obicei, rețelele cu un singur etaj sînt controlate cu un singur flux de instrucțiuni. În acest caz se emite în masiv către toate procesoarele un cîmp de control unic cu $\log_2 m$ biți. Există unele excepții și s-a arătat, folosindu-se sistemul RPA (Jesshope et al 1986), că posibilitatea procesorului de a programa comutatorul, pe baza unor condiții locale, este o extensie foarte puternică a modelului SIMD. Sistemul RPA este descris cu unele detalii în § 3.5.4.

Pentru comutatorul în cruce, cu N intrări, sînt necesari N^2 biți de control, cîte unul pentru fiecare punct de intersecție. Pentru a stabili o conexiune între intrarea i și ieșirea j, este suficient să se modifice corespunzător bitul din poziția i, j a matricei punctelor de intersecție. Dacă sînt acceptate numai corespondențe bijective, atunci la fiecare linie de intrare se poate conecta numai o coloană de ieșire. În acest mod, numărul biților de control poate fi redus la N logN, prin codificarea numerelor coloanelor.

Situația nu este chiar așa de simplă și probabil nu este încă nici bine înțeleasă dacă se consideră rețelele cu mai multe etaje. Acestea necesită tot $O(N\log_2 N)$ biți de control, unde constanta depinde de topologie și de faptul dacă rețeaua este generalizată pentru a putea executa rețelele de

prin folosirea schemelor de memorare asimetrice. Fig. 3.25 prezintă o astfel de schemă pentru o matrice 4×4. Liniile și coloanele pot fi accesate fără conflict, totuși diagonalele rămîn în conflict. Budnick și Kuck (1971), ca și Shapiro (1978) au studiat scheme de adresare asimetrice mai generale, prin considerarea unui număr prim de blocuri de memorie sau, alternativ a unui număr care nu este putere a lui 2. Mai recent, Deb (1980) a prezentat o schemă (4×4) care permite accesul fără conflicte la linii, coloane și ambele diagonale majore. Totuși, nu se elimină conflictele pentru accesul la toate diagonalele circulare (Jesshope 1980 b).

Adresa				
3	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$
2	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$
1	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$
0	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$
Bloc memorie	0	1	2	3

Fig. 3.25 Schemă de memorare asimetrică care permite accesul fără conflict atît la liniile, cît și la coloanele unei matrice 4×4.

Firma Burroughs a proiectat memoria sistemului nenorocos BSP (vezi § 3.4.3) pe baza schemelor asimetrice propuse de Budnick și Kuck (1971). Caracteristica principală a acestei organizări este o memorie cu un număr prim de blocuri de memorie, care permite accesul fără conflicte la toate sub-masivele liniare definite de o adresă de start și distanța de salt. Excepția intervine cînd distanța de salt și numărul blocurilor de memorie au un factor comun. Se întîmplă cînd distanța de salt este un multiplu al numărului de blocuri de memorie, care este un număr prim.

Vom ilustra acest mod de organizare cu masivul nostru 4×4, ca un exemplu. În loc de a avea 4 blocuri de memorie pentru masivul cu 4 procesoare, vom alege numărul prim care urmează, în acest caz, 5. În general, pentru a obține paralelismul N de la un anumit mod de organizare a memoriei vom alege un număr prim M, astfel încît $M \geq N$. Adresa fiecărui element din masiv este dată de ecuația (3.34), unde a este adresa liniară corespunzătoare a elementului.

$$\text{Blocul de memorie:} \quad \mu = |a|_M$$

$$\text{Adresa în bloc:} \quad i = \lfloor a/N \rfloor \qquad (3.34)$$

unde $\lfloor f \rfloor$ furnizează valoarea întreagă a lui f, iar $|f|_g$, valoarea lui f modulo g. Tab. 3.3 prezintă adresele pentru exemplul nostru 4×4, iar fig. 3.26 ilustrează acest mod de memorare. Probabil este un exemplu prost, deoarece accesul la elementele diagonalei principale cauzează conflict. Pentru o matrice $N \times N$ diagonala principală are o distanță de salt $N+1$,5 în exemplul nostru, egal cu numărul blocurilor de memorie. Totuși celelalte sub-masive liniare pot fi accesate fără conflict; liniile, coloanele și diagonala secundară. Ca exemplu să considerăm accesul la linia 2. Adresa de start este 1, iar distanța de salt 4, care definește adresa liniară a fiecă-

rui element din linie. Astfel :

$$\mathbf{a} = (1, 5, 9, 13)$$

şi folosind (3.34)

$$\boldsymbol{\mu} = (1, 0, 4, 3)$$

şi corespunzător

$$\mathbf{i} = (0, 1, 2, 3)$$

Tabelul 3.3. **Măsurarea unui masiv 4×4**

Elementul din A	Adresa liniară	Blocul de memorie	Adresa în bloc
1,1	0	0	0
2,1	1	1	0
3,1	2	2	0
4,1	3	3	0
1,2	4	4	1
2,2	5	0	1
3,2	6	1	1
4,2	7	2	1
1,3	8	3	2
2,3	9	4	2
3,3	10	0	2
4,3	11	1	2

La accesul unui sub-masiv, $\boldsymbol{\mu}$ defineşte un vector de mapare sau permutare ce poate fi folosit pentru programarea reţelei de comutare, astfel că blocurile de memorie să fie conectate corect la procesoare. Vectorul de indexare **i** este folosit pentru modificarea adresei de bază. Trebuie notat că elementele lui i corespund blocurilor de memorie date în $\boldsymbol{\mu}$ şi trebuie deci permutate conform lui $\boldsymbol{\mu}$ pentru obţinerea indexului în blocul de memorie. O privire la fig. 3.26 va confirma că vectorii anteriori selectează într-adevăr linia a 2-a a matricei.

3.4. O perspectivă istorică

3.4.1. O istorie amalgamată

Paralelismul poate fi aplicat fie biţilor unui cuvînt, fie unui număr de cuvinte, sau atît biţilor, cît şi cuvintelor. Acest compromis a fost deja discutat în § 3.2.3. Alegerea soluţiei paralel pe bit/serial pe cuvînt, de către Von Neumann şi alţii în anii 1950 a stabilit o direcţie în ce priveşte arhitecturile de calculatoare care mai recent a început să fie contestată în mod serios. Alegerea a reflectat corect restricţiile tehnologice şi de imple-

mentare din momentul respectiv, cînd circuitele logice erau foarte rapide și costisitoare, iar memoriile erau relativ lente și necostisitoare. Aceste restricții au determinat proiectarea structurii procesor-memorie, orientată pe cuvînt (așa-numita arhitectură Von Neumann), care menținea părțile costisitoare ocupate (de ex. procesoarele construite cu tuburi).

Adresa					
3	$A_{4,4}$	X	$A_{1,4}$	$A_{2,4}$	$A_{3,4}$
2	$A_{3,3}$	$A_{4,3}$	X	$A_{1,3}$	$A_{2,3}$
1	$A_{2,2}$	$A_{3,2}$	$A_{4,2}$	X	$A_{1,2}$
0	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{4,1}$	X
Bloc memorie	0	1	2	3	4

Fig. 3.26 Schemă de memorare care folosește un număr prim de blocuri de memorie (cinci), care permite accesul fără conflict la toate sub-masivele liniare (linii, coloane etc.) ale unei matrice 4×4, cînd elementele succesive nu sînt separate prin cinci poziții.

La calculatoarele moderne, atît memoria, cît și circuitele logice sînt ieftine, iar componenta critică a devenit firul de interconectare. În mod obișnuit multe sute de mii de tranzistori intră în structura unui circuit VLSI, iar în viitorul apropiat numărul lor va crește probabil la milioane. În mod sigur s-au produs deja circuite cu mai mult de un milion de tranzistoare (vezi cap. 6). Deoarece memoria și circuitele logice pot fi construite acum cu aceeași tehnologie, vitezele și costurile sînt comparabile; într-adevăr, este dezirabil de a integra atît memoria cît și circuitele logice pe același circuit. Acești factori tehnologici mai solicită arhitecturi cu o echilibrare între partea de memorie și cea de prelucrare și, mai mult, să fie integrate mult mai apropiat componentele care în arhitectura Von Neumann sînt separate.

Masivul de procesoare și într-o oarecare măsură sistemul multiprocesor reprezintă o cale alternativă de evoluție față de procesorul Von Neumann convențional, prin aceea că mai multe procesoare cooperează la o singură problemă, fiecare accesînd date din memoria proprie. Trebuie observat că un comutator care realizează un set complet de permutări între procesoare și memorii determină ca noțiunea de proprietate să aibă o natură temporară! Probabil calculatorul SIMD cu conexiuni după două direcții ortogonale a fost primul rival cu această arhitectură, cum se va vedea mai departe. Totuși, nici unul din aceste sisteme nu a fost un succes comercial. Unele au avut succes tehnic, iar unele au fost comercializate, dar preferința utilizatorului pentru calculatoare care să execute FORTRAN serial standard a dominat piața supercalculatoarelor. Prin creșterea interesului pentru paralelism, dezvoltarea limbajelor paralele (vezi cap. 4) și creșterea raportului performanțe/cost care poate fi obținut cu circuitele VLSI multiplicate, masivele de procesare și sistemele multiprocesor încep să se răspîndească.

(i) *Un start timpuriu*

Ideea unor calculatoare conectate, care ar putea fi folosite la rezolvarea unor probleme spațiale, a fost concepută în 1958 în lucrarea lui Unger (1958). La această mașină, un masiv bidimensional de module logice conectate cu vecini era comandat de un singur controler master. Fiecare modul (azi s-ar chema element de prelucrare sau PE) constă dintr-un

acumulator, un procesor boolean și 6 biți RAM. Data intră în modul fie prin conectarea modulelor sub forma unui registru de deplasare, fie direct de la controlerul master. Controlerul funcționa ca un secvențiator Von Neumann convențional și putea, cu o excepție, să facă un salt funcție de suma logică a datelor din toate modulele masivului (acumulatoarele furnizau aceste date). Excepția reprezenta una din cele mai importante inovații ale lui Unger, ce poate fi întîlnită azi la aproape toate masivele de procesoare.

Această posibilitate reprezenta singurul salt dependent de date, posibil în proiectul lui Unger, care s-ar traduce cu expresiile paralele :

```
FOR ALL PROCESSORS
    IF ANY (ACCUMULATOR=TRUE)
        THEN ACTION 1
        ELSE ACTION 2
```

Desigur cu o modificare a semnificației logice, s-ar putea obține și un salt pentru condiția „toate acumulatoarele adevărate". Această arhitectură nu a fost implementată niciodată, datorită „cifrelor alarmante" estimate de Unger pentru numărul de porți necesare. Cifrele actuale sînt de aproximativ zeci de mii de porți logice, ce pot fi implementate cu ușurință azi într-un singur circuit.

(ii) *SOLOMON*

Proiectul calculatorului SOLOMON (Slotnick et al 1962, Gregory și Mc. Reynolds 1963) constă dintr-un masiv de 32×32 PE și deși nu a fost construit niciodată, a fost precursorul seriei de proiecte ILLIAC (University of Illinois Advanced Computer), care a culminat cu ILLIAC IV și Burroughs BSP (vezi §1.1.4. și 3.4.3.). SOLOMON a avut o influență majoră și asupra sistemului ICL DAP (vezi §3.4.2), fiind important pentru introducerea unui alt concept de control al masivelor de procesoare SIMD, acela al controlului de mod (sau activitate, cum s-ar chema azi). Modul unui PE era un indicator pe un bit, care putea fi setat funcție de valoarea unor date locale și folosit apoi pentru determinarea acțiunii instrucțiunilor următoare. În particular, putea fi folosit la inhibarea memorării rezultatelor, deci pentru asigurarea execuției unei operații condiționate local.

Controlul modului sau al activității este echivalent cu următoarele instrucțiuni :

```
FOR ANY PROCESSOR
    IF MODE
        THEN ACTION 1
```

Construcția paralelă IF-THEN poate fi scrisă uneori ca

```
WHERE MODE
    THEN ACTION 1
```

(iii) *ILLIAC IV*

ILLIAC IV a fost cu siguranţă primul masiv de procesoare construit cu succes tehnic, de care îşi aminteşte cu plăcere cel puţin unul din autori, deoarece a fost folosit la începutul unui program de cercetări privind folosirea calculatoarelor paralele la rezolvarea unor probleme ştiinţifice şi inginereşti (Jesshope şi Craigie 1980). Pentru mulţi cercetători sistemul ILLIAC IV instalat la Ames de NASA în 1972, a oferit un serviciu complet via ARPANET, începînd cu 1975.

ILLIAC IV era constituit din 64 procesoare complexe aranjate după două direcţii ortogonale. Procesoarele lucrau pe 64 de biţi deşi puteau fi reconfigurate sub forma unui masiv cu dimensiunea de 256 pentru a lucra cu date pe 8 biţi. Proiectul iniţial constă din 4 sferturi, fiecare cu dimensiunea masivului construit acum iar întregul, ce folosea 4 controlere, ar fi fost capabil să realizeze mai mult de 1 Gflop/s (10^9). Una din caracteristicile originale ale lui ILLIAC IV, posibil de realizat numai datorită complexităţii procesorului, a fost posibilitatea execuţiei locale a operaţiilor cu indecşi. În fiecare procesor putea fi folosit un registru X pentru modificarea adresei emise de controlerul central. Astfel, fiecare procesor poate accesa o locaţie diferită în memoria propriu-zisă.

Această posibilitate poate fi exprimată :

FOR ALL PROCESSORS {LABELLED I=[O FOR 64]}

A[I] : =MEMORY [X[I]]

Aşa cum s-a arătat în §3.3.8, această posibilitate este foarte puternică, în special cînd se accesează masive după mai multe dimensiuni.

Deşi ILLIAC IV a fost livrat cu întîrziere, a depăşit bugetul, nu a atins niciodată performanţele specificate şi a fost foarte puţin fiabil, a asigurat comunităţii cercetătorilor din domeniul masivelor paralele un stimul imens. Este suficient să trecem în revistă limbajele paralele dezvoltate pentru ILLIAC. De exemplu, TRANQUIL (Abel et al 1969), asemănător FORTRAN-ului şi ACTUS (Perott 1979) asemănător PASCAL-ului, au fost concepute iniţial pentru ILLIAC IV.

(iv) *Adevăratul început*

Anii '70 au înregistrat dezvoltarea multor alte proiecte de masive de procesoare. În Marea Britanie s-au fabricat CLIP, la University College din Londra, special proiectat pentru prelucrarea imaginilor (Fountain şi Goetcherian 1980) şi ICL DAP, descris mai pe larg în §3.4.2.. Primul sistem ICL DAP a fost livrat în 1979 la Queen Mary College din Londra (QMC) şi deşi a fost bine primit de comunităţile de cercetători, nu a reprezentat un succes comercial pentru ICL. Se pare că succesorul lui, un DAP LSI va avea o soartă similară ; oricum, în momentul redactării acestei lucrări, ICL a transferat dezvoltarea sistemelor DAP unei firme nou create denumite Active Memory Technology Ltd (AMT Ltd). Urmează să vedem dacă această companie nouă va fi capabilă să concureze cu numărul în creştere de produse cu transputere.

Din cele 6 sisteme DAP construite, unul a mers la QMC, două la Edinburgh University, unul la National Phisical Laboratory din Teddington, iar ultimul a fost folosit de către companie pentru proiectarea circuite-

lor integrate (CAD). În Statele Unite s-au construit, în aceeași perioadă, alte masive de procesoare STARAN și PEPE. STARAN este un procesor asociativ construit de Goodyear Aerospace, iar PEPE a fost proiectat pentru combaterea rachetelor balistice.

Cele mai multe din sistemele amintite mai sus au avut succesori, dintre care unii vor fi descriși în secțiuni ulterioare ale acestui capitol. De exemplu, sistemul BSP, construit de Burroughs, a rezultat din experiența firmei de la contractul pentru realizarea lui ILLIAC IV. Deși s-a construit o singură mașină BSP, ea este descrisă în §3.4.3, deoarece a reprezentat la acea dată cel mai reprezentativ masiv de procesoare. Ea a avut de concurat cu procesoarele vectoriale, printre care și cu CRAY-1. De aceea ar fi trebuit să fie capabilă să depășească performanța maximă a lui CRAY-1 ; cînd, de fapt, nu realiza decît o fracție din aceasta. Faptul că, spre deosebire de CRAY, BSP a fost proiectat să realizeze un procent mare din performanța maximă pentru o gamă largă de probleme programate în FORTRAN, nu a impresionat nici utilizatorii, nici reprezentanții lor. Probabil de aici se poate extrage o morală.

3.4.2. ICL DAP

Proiectarea sistemului DAP (distributed array processor) pilot a început în 1974, urmînd proiectul SOLOMON (Gregory și McReynolds 1963), care consta dintr-un masiv bi-dimensional de 1024 procesoare pe 1 bit. Totuși, DAP a introdus două contribuții noi la formula SOLOMON. Prima a fost o caracteristică hardware care consta în secționarea masivului după două direcții ortogonale. Unitatea de control master (MCU) avea un număr de registre pentru fiecare direcție a masivului DAP. Se foloseau două magistrale de date ortogonale, care străbăteau liniile și coloanele de elemente procesoare (PE). Aceste magistrale aveau cîte 1 bit pentru fiecare bit al registrului MCU, care parcurgea fie o linie, fie o coloană din masiv. Astfel $PE_{i,j}$ va avea o magistrală pe 1 bit direct la biții i și j ai registrului MCU. Aceste magistrale colectează și emit date unor secțiuni (felii-slices) din masivul DAP, asigurînd astfel flexibilitate în manipularea datelor.

A doua contribuție se referă la modul în care DAP este integrat intr-un sistem complet. DAP este proiectat să emuleze un modul de memorie al unui calculator de uz general ICL și să prelucreze autonom date intr-o manieră paralelă. Acest concept i-a dat și numele, deoarece puterea de calcul este distribuită în memoria unui calculator convențional.

Sistemul pilot a fost terminat în mai puțin de doi ani de la începere, ca un masiv de 32 × 32 PE cu 1Kb de memorie fiecare. După 6 ani, în 1980, s-au livrat primele trei bucăți beneficiarilor. Acestea erau masive de 4096 PE,, organizate în aceeași geometrie, fiecare procesor cu 4 Kb de memorie. Se obține un total de 2 MB (ulterior crescută la 8 MB), atașat la una din mașinile cele mai performante din gama ICL 2900. Între pilot și mașinile comerciale există unele diferențe. Aici, vom descrie prima versiune produsă. Cititorul interesat poate afla detalii despre mașina pilot in lucrarea lui Reddaway (1973), iar aspecte privind evaluarea ei pentru

În sfîrşit, magistrala coloană asigură calea pentru extragerea instrucţiunilor DAP de către MCU. Instrucţiunile DAP sînt memorate cîte două pe o linie, iar o linie este citită din memorie într-o perioadă de ceas. În anumite condiţii instrucţiunile pot fi memorate de buferul pentru instrucţiuni, pentru execuţie repetată. Mai multe detalii se vor da ulterior.

Se mai poate vedea în fig. 3.29 că magistrala linie conectează masivul DAP la registrele MCU după direcţia ortogonală cu prima. Această magistrală are 1 bit pentru fiecare linie de procesoare din masiv şi este folosită în exclusivitate pentru transmiterea datelor la sau de la registrele MCU. Fig. 3.30 prezintă diversele componente şi căi de date care constituie un singur element procesor. Masivul formează o grilă bi-dimensională, fiecare procesor aflîndu-se într-un nod şi avînd 4 vecini. Conexiunile de la marginea masivului sînt definite de instrucţiunea în curs de execuţie. Geometria unei instrucţiuni este fie planară, fie ciclică în linii şi coloane. Geometria planară defineşte o intrare zero la margini, în timp ce geometria ciclică asigură conexiuni periodice în linii sau coloane ale masivului. Geometria liniilor şi coloanelor poate fi stabilită independent.

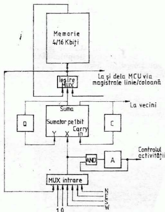


Fig. 3.30 O diagramă simplificată a elementului procesor ICL DAP.

Într-un procesor există trei registre pe 1 bit, două multiplexoare şi un sumator complet pe 1 bit. Registrul A asigură controlul programat asupra acţiunii elementului procesor, deoarece anumite instrucţiuni de memorare sînt executabile dacă acest registru de activitate este setat. Registrul A are intrarea comandată de porţi şi care permit realizarea de funcţii „şi" între registru şi intrare, realizînd astfel rapid măşti de control. Celelalte registre sînt acumulatorul (Q) şi registrul de memorare a transportului (C).

Sumatorul adună Q, C şi data de la intrarea elementului procesor, producînd suma şi transportul, care pot fi memorate în registrul Q, respectiv C. O excepţie este reprezentată de situaţia cînd se execută o adunare cu un operand aflat în memorie. Această familie de instrucţiuni consumă 1,5 perioade, în cursul căora se citeşte operandul din memorie şi se scrie bitul sumă produs în aceeaşi locaţie (cum se va vedea în continuare, cele mai multe instrucţiuni se execută într-o perioadă de ceas), salvînd astfel o jumătate de perioadă faţă de o adunare cu acumulator, urmată de transferul rezultatului în memorie.

Sînt incluse şi elemente pentru calculul parităţii (nu apar în figuri). Acestea verifică atît funcţionarea memoriei, cît şi modul de execuţie a funcţiilor logice (Hunt 1978). Cînd lucrează ca memorie a calculatorului 2900, se foloseşte codul Hamming complet , care permite corecţia erorilor unice şi detecţia erorilor duble pentru fiecare dată citită de 64 biţi.

DAP execută instrucţiunile în două faze, în ciclii de extragere şi execuţie. Fiecare durează o perioadă de 200 ns. Cînd o instrucţiune intervine după şi în interiorul unei instrucţiuni hardware specială, de tipul DO cu ciclare, extragerea instrucţiunilor se execută o singură dată pentru cele N iteraţii ale buclei. Această instrucţiune are două cîmpuri de date, un cîmp care indică lungimea buclei şi un cîmp pentru numărul de execuţie al buclei. Lungimea maximă este de 60 instrucţiuni, iar numărul maxim de execuţie este de 254. În interiorul unei bucle, instrucţiunile pot avea adresele incrementate sau decrementate cu 1 la fiecare pas. Bucla DO este esenţială pentru construirea programelor ce prelucrează cuvinte. Instrucţiunile se execută cu rata de una la fiecare 1,5 perioade de ceas, cînd extragerea are loc la fiecare execuţie (două instrucţiuni sînt extrase într-un ciclu).

Cele mai multe instrucţiuni DAP au formatul din fig. 3.31 (a). Codul operaţiei şi cîmpul de inversare specifică efectiv tipul instrucţiunii. Bitul de inversare creează perechi de instrucţiuni identice, cu excepţia faptului că una din intrări este inversată. De exemplu, QA şi QAN sînt perechea care încarcă registrul Q cu ceea ce se află în registrul A. QAN inversează intrarea. Multe instrucţiuni DAP au astfel de perechi complementare. Celelalte două cîmpuri de 1 bit menţionează dacă adresa este incrementată sau decrementată, în cadrul unei bucle DO.

Cele două cîmpuri cu 3 biţi specifică registrele MCU. Primul este setat dacă este necesar un registru pentru date, iar al doilea dacă este necesară modificarea instrucţiunii. Cele două cîmpuri rămase reprezintă fie o adresă de memorie sau un deplasament efectiv, în cadrul unei instrucţiuni de deplasare. Aceste cîmpuri pot fi modificate de registrul MCU definit de cîmpul modificator.

O instrucţiune care referă memoria conţine două cîmpuri de adresă pe 7 biţi. Unul este numărul unei linii sau coloane, iar al doilea un deplasament pe 7 biţi. Aceste adrese se adună la conţinutul registrului modificat (fig. 3.31. (b)) pentru a forma adresa absolută. Este permis un transport la calculul adresei numai dacă sînt referite linii. Pentru instrucţiunile care adresează coloane, numărul coloanei este trunchiat la 6 biţi, pentru a se obţine o valoare modulo 64.

În cadrul instrucţiunilor care deplasează datele prin masiv, cele două cîmpuri de adresă menţionează o adresă relativă în masiv. Un cîmp defineşte geometria şi direcţia deplasării, iar celălalt este folosit ca numărător pentru deplasare. Ambele pot fi modificate cu ceea ce se află într-un registru MCU cu formatul din fig. 3.31 (c). Valorile posibile ale direcţiei sînt, identice, N, S, E, şi V. Geometria are patru variante, după linii şi coloane, programate independent în plan sau ciclic.

Indiferent de modul de adresare, dacă cîmpul modificator este zero, nu se produce nici o modificare.

Nu încercăm să prezentăm aici toate instrucțiunile, cititorul interesat având posibilitatea să consulte manualul limbajului de asamblare DAP APAL (ICL 1979 c).

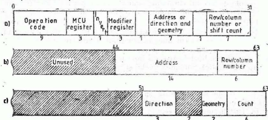


Fig. 3.31. Formatele instrucțiunii și a modificatorului : (a) instrucțiune ; (b) modificatorul adresei ; (c) modificatorul deplasării.

Vom încerca, totuși să sistematizăm instrucțiunile în grupe, în tab. 3.4, și să indicăm căile de date conceptuale create de aceste instrucțiuni, în fig. 3.32. Liniile întrerupte indică controlul exercitat de registrul de activitate.

Tabelul 3.4 **Sumar de instrucțiuni DAP.**

Registru-registru	Registru-memorie
adunare pe 1 bit	adunare de un bit la locație
Adunare completă sau pe jumătate	Adunare completă sau pe jumătate
adună la Q	adună la conținutul locației
transport la C	transport la C
Adunare vectorială	
adunare serială	
$Q \leftarrow Q + C$	
Transmisie	Încărcare/memorare
registru-la-registru	Încărcare și memorare a registrelor Q și A (linii,
în interiorul PE, inclusiv instrucțiuni de deplasare	coloane sau întreg planul de memorie MCU/ memorie
MCU/masiv	
Încarcă registrele MCU, emisie sau scriere selectivă la/de la Q și A	Încarcă registrele MCU, emisie sau scriere selectivă la/de la memorie
	numai MCU
	Control operații logice sau aritmetice cu registrele MCU

Modul normal de funcționare a calculatorului DAP este serial pe bit, în paralel pe 4096 cuvinte. Microcodul pentru adresare va folosi posibilitatea hardware de execuție a buclelor DO, prin accesul consecutiv la biții de date, memorați continuu în memorie.

Alternativ, elementele de calcul DAP pot fi configurate pentru a forma un sumator paralel sau cu transport anticipat (fig. 3.3) (PE pot fi legate în orice direcţie). Astfel pot fi prelucrate în paralel 64 cuvinte a 64 biţi. Transportul este propagat cu cel puţin 4 poziţii binare într-o perioadă de ceas.

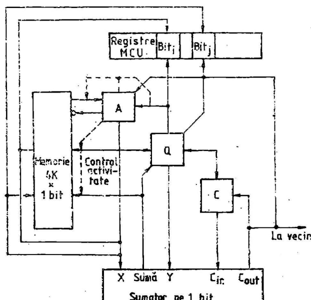


Fig. 3.32. Căile de date conceptuale într-un element de procesoare DAP.

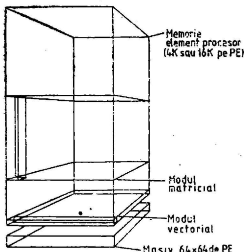


Fig. 3.33 O diagramă a memoriei şi masivului de procesoare DAP, care prezintă cele două moduri posibile de memorare; modul vectorial, cînd cuvîntul este memorat orizontal, într-o linie de PE; modul matricial, cînd cuvîntul este memorat vertical într-un singur PE.

Fig. 3.33 prezintă structurile de date pentru aceste moduri de lucru, modul matricial şi modul vectorial. Prelucrarea cuvintelor în modul vectorial se execută întotdeauna cu un cuvînt pe linie, chiar dacă este o risipă

de elemente procesoare pentru numerele mici (mai mici ca 64 la calculatorul DAP). RPA, descris în § 3.5.4., este proiectat să folosească acestei elemente neutilizate, asigurind o arhitectură mai flexibilă. Trebuie să remarcăm că atunci cînd datele nu corespund structurii calculatorului DAP, înaintea execuției calculelor, datele trebuie reorganizate, ceea ce înseamnă o pierdere şi de spaţiu de memorie şi de putere de calcul.

La nivelul sistemului DAP, care este strîns integrat sistemului gazdă (ICL 2900), codul DAP îi apare utilizatorului ca o parte a unui limbaj mixt, cu subrutine DAP FORTRAN, apelate din programele FORTRAN executate pe 2900. Ca un exemplu, să considerăm aplicaţia tipică din fig. 3.34.

Se poate vedea că gazda asigură serviciile sistem, inclusiv compilarea, introducerea datelor şi analiza. Un singur punct de intrare din programul FORTRAN gazdă va transfera controlul subrutinelor DAP între DAP şi gazdă, comunicaţia se realizează prin *blocuri comune* care, cînd sînt accesate de DAP, vor fi încărcate în memoria 2900 corespunzătoare spaţiului de adresare DAP, de unde sînt accesibile ambelor sisteme.

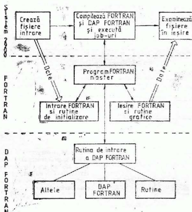


Fig. 3.34 Circulaţia datelor şi a comenzilor în cadrul programelor DAP.

Limbajul DAP FORTRAN, un limbaj paralel FORTRAN, este prezentat în § 4.4.2. Limbajul de asamblare DAP, APAL poate fi interfaţat atît cu FORTRAN, cît şi DAP FORTRAN. Cînd se execută operaţii în virgulă mobilă sau rutine sistem standard (înalt optimizate), utilizarea APAL nu produce cîştiguri substanţiale. Totuşi, dacă pot fi găsiţi algo-

ritmi care să exploateze natura liniară a elementelor procesoare, se pot obţine, prin codificarea în limbaj de asamblare, creşteri de performanţe de mai multe ordine de mărime (Eastwoad şi Jesshope 1977).

Cum elementele procesoare operează serial, toate operaţiile aritmetice trebuie definite prin program. Pentru modul de prelucrare al masivului DAP, operaţiile aritmetice se vor executa ca secvenţe de operaţii pe bit. Aceasta înseamnă că va fi o dependenţă puternică a performanţei de lungimea cuvîntului. Pentru operaţiile cu întregi, proporţionalitatea este cu lungimea cuvîntului, în timp ce pentru adunare şi înmulţire, cu pătratul lungimii cuvîntului. În cazul calculelor în virgulă mobilă, aceste proporţionalităţi sînt într-un fel mascate de overhead-urile necesare pentru manipularea exponentului şi a mantisei. Totuşi, în ambele cazuri, dependenţa de lungimea cuvîntului reprezintă aspectul pozitiv, dar şi negativ al performanţei sistemului DAP. Aspectul negativ constă în aceea că aplicaţiile care impun calcule de mare precizie se execută lent pe DAP; în schimb, în cazul aplicaţiilor care folosesc cuvinte scurte, se pot executa mai multe sute de milioane de operaţii pe secundă.

Pentru cele mai multe probleme, calculele în virgulă mobilă pe 32 de biţi oferă o precizie suficientă, de aceea furnizăm în tab. 3.5 timpii de execuţie corespunzători. Aceştia sînt luaţi din lucrarea lui Reddaway 1979) şi se referă la rutinele sistem optimizate. Aceste valori pot fi folosite pentru evaluarea timpilor de execuţie ai programelor DAP FORTRAN, deoarece overheadurile de manipulare în cadrul limbajului de nivel înalt sînt minime. Acest lucru se datorează vitezei mari cu care poate DAP să manipuleze datele, lucru ilustrat în tab. 3.5 de timpii de execuţie pentru atribuire.

Tabelul 3.5, **Rutine aritmetice DAP (reprezentare pe 32 biţi) X, Y şi Z sînt reale (4096 elemente); IX; IY şi IZ sînt întregi (4096 elemente); S este un scalar real.**

Operaţia	Timp (μs)	Viteza de calcul(Mop/s)
Z ← X	17	241
Z ← X − S	40−130	32−102
Z ← X^2	125	33
Z ← X + Y	150	27
Z ← SQRT(X)	170	24
Z ← X × Y	250	16
Z ← LOG(X)	285	14
Z ← X/Y	330	12
Z ← MAX (X, Y)	33	124
Z ← MOD(Z)	1	4096
IZ ← IX + IY	22	186
S ← SUM(X)	280	175
S ← MAX (X)	48	85

Unele din rutinele aritmetice au fost evaluate în cadrul unor programe DAP FORTRAN, pentru a se stabili cît de mare este overhead-ul pentru un limbaj de nivel înalt. Aceşti timpi sînt prezentaţi în tab. 3.6 Se poate vedea că variază de la mai puţin de 10% pentru operaţii cu o matrice

(4096 elemente) la ceva peste 20% pentru buclele DO care execută 10 operaţii matriciale (40960 elemente).

Dacă se folosesc algoritmi care să lucreze la nivel de bit, se pot face economii substanţiale. Unele valori din tab. 3.5 ilustrează această observaţie, şi anume unde operaţiile furnizează viteze de execuţie

Tabelul 3.6 **Timpul de execuţie pentru DAP FORTRAN (reprezentare pe 32 biţi). X, Y şi Z sînt reale (4096 sau 40960 elemente) S este un scalar real.**

Operaţie	4096 elemente		40960 elemente	
	Timp (μs)	Viteză (Mflop/s)	Timp(μs)	Viteză (Mflop/s)
Z ← X + Y	152	27	1848	22
Z ← X × Y	272	15	3048	13
Z ← X − S*	112−200	20−37	1368−2272	18−50
Z ← X²	152	27	1816	23
Z ← SQRT(X)	192	21	2208	19*
Z ← X/Y	376	11	4080	10

* Se foloseşte S = 2 şi S = e pentru minim şi valori tipice

surprinzător de contradictorii. Probabil, cea mai uimitoare este suma elementelor unei matrici cu 4096 elemente. Ne putem aştepta ca producerea rezultatului să consume aproximativ 12 adunări în virgulă mobilă ($\log_2 4096 = 12$); de fapt se consumă mai puţin de 2! Alte exemple sînt extragerea radicalului şi logaritmului care, în mod normal, se execută, iterativ prin multe înmulţiri în virgulă mobilă. Unii din algoritmii folosiţi pentru aceste funcţii sînt descrişi în lucrările lui Flanders et al (1977) şi Gostick (1975).

3.4.3. Burroughs BSP

Spre deosebire de DAP, BSP este un masiv de procesoare complexe, capabile să execute operaţii în virgulă mobilă cu numere reprezentate pe 48 de biţi. Foloseşte din plin experienţa acumulată de Burroughs în calitatea de participant la proiectarea şi construirea lui ILLIAC IV. Deşi proiectul are multe caracteristici interesante, a suferit din cauza unor reorganizări interne ale firmei Burroughs, nefiind niciodată lansat în producţie. Probabil aceasta a fost o decizie bună, deoarece maşina nu concurează performanţele calculatorului CRAY-1. De fapt nu concurează nici predecesorul, ILLIAC IV, cu viteză maximă de 50 Mflop/s în comparaţie cu 80−100 posibil la ILLIAC IV. În ciuda acestei valori dezamăgitoare, proiectul a înlăturat un număr de deficienţe majore ale predecesorilor săi şi prezintă un interes suficient pentru a-l include în lucrarea prezentă.

Una din problemele importante atacate de Burroughs a fost cea a menţinerii unei baze de date mari, care să ducă la folosirea continuă a masivului de procesoare. Aceasta a fost o limitare serioasă la ILLIAC IV (Feierbach şi Stevenson 1979 a) care posedă numai 128 Kw memorie RAM şi un spaţiu mare de memorare pe discuri. În contrast, BSP are memorie

RAM de la 1 la 8 Mw, plus o memorie secundară electronică rapidă. BSP are organizată memoria astfel incit multe submulţimi regulate pot fi accesate în paralel, fără conflict.

Alte învăţăminte căpătate de Burroughs de la experienţa cu ILLIAC IV se referă la organizarea procesorului de control, un alt punct slab la

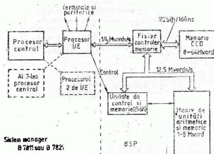


Fig. 3.35 Legătura dintre BSP şi calculatorul gazdă B7800.

ILLIAC IV. În timp ce ILLIAC IV avea o memorie tampon mică şi numai posibilităţi de prelucrare limitate, BSP asigură procesorului de control 256 Kw şi un procesor scalar mai complex. Această memorie este folosită pentru programe şi date.

Detaliile privind sistemul BSP provin de la prototipul construit şi testat în 1980 (Burroughs 1977 a-d, Augustin 1979). Fig. 3.35 prezintă configuraţia sistemului BSP.

Interfaţa dintre sistemul manager şi BSP asigură două căi de date, una lentă şi una rapidă. Calea lentă (500 KB/s) interfaţează procesorul de I/E direct cu unitatea de control BSP, fiind folosită pentru trecerea mesajelor şi a informaţiei de control între cele două sisteme. A doua cale (1/4 Mw/s) interfaţează procesorul de I/E cu controlerul memoriei secundare, fiind folosită pentru transmiterea programului şi datelor pentru execuţie pe BSP. Memoria secundară reprezintă o interfaţă bufer între sistemele front-end şi back-end, pentru comunicaţii cu o lărgime mare a benzii.

Memoria secundară este una din cele trei componente importante ale calculatorului BSP. Celelalte două: procesorul de control şi masivul de procesoare apar în fig. 3.36 şi sînt descrise în continuare.

Procesorul de control conţine patru unităţi asincrone care asigură controlul masivului, planificarea lucrărilor, operaţii de I/E şi gestiunea fişierelor, detectarea erorilor şi, în sfîrşit, comunicarea comenzilor între sistemul gazdă şi BSP.

Procesorul de control are 256 Kw de memorie MOS cu un ciclu de 160 ns. Această memorie păstra atît instrucţiunile scalare vectoriale, cît şi datele scalare, fiind conectată la memoria secundară.

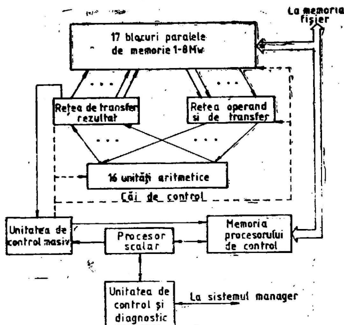


Fig. 3.36 Procesorul de control BSP şi secţiunea masiv, cu componentele majore şi căile de date

Unitatea de prelucrare a scalarilor era un procesor convenţional, orientat registru, care folosea hardware identic cu cel întîlnit la cele 16 unităţi aritmetice (AU) din masiv. Totuşi, diferă prin aceea că avea un procesor propriu pentru instrucţiuni, care citea şi decodifica instrucţiuni memorate în memoria de control a procesorului.

Procesorul de control avea 16 registre de 48 biţi şi lucra cu o perioadă a ceasului de 80 ns. Executa atît operaţii numerice cît şi ne-numerice, cele în virgulă mobilă la o viteză maximă de 1,5 Mflop/s. Una din sarcinile importante ale procesorului scalar este preprocesarea instrucţiunilor vectoriale. Se includ aici optimizările, verificarea hazardelor vectoriale, ca şi inserarea diverselor cîmpuri de informaţii în formatul instrucţiunii vectoriale. Pentru execuţia acestor operaţii se foloseau bufere vectoriale de 120 biţi, cîte unul pentru un descriptor de instrucţiune vectorială. Instrucţiunile gata de execuţie erau trecute unităţii de control a masivului care le introducea într-o coadă de aşteptare, le decodifica şi transmitea microcodul de control secţiunii de masiv a calculatorului BSP.

Trebuie să subliniem că procesorul de control avea o funcţionare complet independentă de secţiunea masivului. Odată iniţializată, coada cu instrucţiuni ar fi trebuit să alimenteze continuu secţiunea de calcul a masivului. Procesorul scalar prelucra independent instrucţiuni scalare şi pre-procesa instrucţiuni vectoriale. Această soluţie a fost folosită prima

dată la ILLIAC IV, unde posibilitatea suprapunerii funcționării celor două secțiuni trebuia programată cu multă atenție. La BSP, procesorul scalar și unitatea de control a masivului asigura operațiile necesare în momentul execuției.

Secțiunea masivului conținea patru unități care lucrau ciclic sub forma unui pipeline cu 5 elemente. Operațiile sau task-urile executate de acest pipeline sînt :

(a) citește operanzii din memoria paralelă;
(b) aliniază operanzii corespunzători unităților aritmetice;
(c) execută operația;
(d) aliniază rezultatul cu blocurile de memorie;
(e) memorează rezultatele în memoria paralelă.

Aceste instrucțiuni se suprapun cu seturi succesive de 16 elemente luate din instrucțiunile vectoriale prelucrate de unitatea de control a masivului. Numărul perioadelor de ceas (160 ns) necesare pentru fiecare din aceste task-uri este variabil funcție de numărul operanzilor și operațiilor executate. Suprapunerea în pipeline este controlată cu fragmente diferite de microcod denumite tipare (templates). Unitățile aritmetice erau de uz general și comandate cu o singură secvență de micro-instrucțiuni. Cuvîntul era pe 100 biți, asigurînd accesul direct la funcțiile primitive ale AU care, în plus, față de operatorii în virgulă mobilă mai aveau și un set comprehensiv de operatori de editare și manipulare a cîmpurilor. Aici sînt incluși și operatori speciali pentru conversia formatului FORTRAN.

Adunarea și înmulțirea în virgulă mobilă se execută în două perioade a 160 ns, de unde pentru un AU viteza de execuție de 3 1/8 Mflop/s. Masivul de 16 realizează în consecință, viteza maximă de 50 Mflop/s. Atît împărțirea cît și extragerea radicalului sînt implementate prin iterații Newton-Raphson și memorii ROM, care furnizau prima aproximare.

Cuvîntul pe 48 biți folosit constant de Burroughs, are 36 biți pentru mantisă, ceea ce înseamnă o precizie de 10—11 cifre zecimale. Exponentul pe 11 biți produce un domeniu de variație de $\pm 10^{+307}$. Acumulatoarele și registrele cu lungime dublă permit implementarea hardware a operațiilor în dublă precizie.

Memoria paralelă a secțiunii de masiv BSP conținea 17 blocuri de memorie, cu perioada de 160 ns. 17 este numărul prim care urmează lui 16 (numărul de procesoare). Această organizare, împreună cu o rețea de interconectare completă asigură accesul fără conflict la toți vectorii liniari dacă incrementul între adresele elementelor din memorie nu este multiplu de 17. Această tehnică de acces a fost descrisă în §3.3.8.

La BSP s-au folosit rețele cu accesuri încrucișate complete cu detecție și corecție de eroare. Și aceste comutatoare lucrează cu perioada de 160 ns.

Instrucțiunile vectoriale prelucrate de unitatea de control a masivului constau din 1 la 4 operații (1 la 5 seturi de operanzi), producînd un singur rezultat vectorial sau scalar. Utilizarea unei astfel de instrucțiuni de nivel înalt permite o optimizare mai bună a utilizării hard-ului, nu numai la nivelul registrelor dar și la maximizarea suprapunerii în pipeline. Se poate înțelege că s-a atins o echilibrare a tuturor elementelor unității pipeline pentru operații triadice, de exemplu :

$$A = B \text{ op } C \text{ op } D$$

Aici, cele patru referinţe la memorie de 160 ns şi cele 2 operaţii (+, — sau *) de 320 ns pot fi suprapuse total. Totuşi cele 2 reţele de transfer, în intrare şi ieşire, utilizează numai 3, respectiv 1 din cele 4 perioade.

BSP a fost proiectat ca un procesor de limbaj de nivel înalt (FORTRAN), care realizează optimizări hardware şi transformări ale codului utilizator în momentul execuţiei. A fost necesară folosirea unui procesor de control complicat pentru a menţine o încărcare continuă a reţelei pipeline :

(a) extragerea operanzilor;

(b) alinierea operanzi;

(c) execuţia operaţie;

(d) alinierea rezultate;

(e) memorarea rezultate.

Din punct de vedere al limbajului FORTRAN, optimizarea utilizării unităţilor aritmetice multiple s-ar realiza cu un set de bucle DO încuibărite. De fapt operaţiile vectoriale BSP folosesc una sau două bucle încuibărite. Totuşi multe bucle actualizează parametrii în momentul execuţiei, de aceea şi formatul descriptor pentru operaţiile vectoriale poate conţine unii parametri actualizaţi în faza de execuţie.

Aceste instrucţiuni maşină au forma unor operaţii vectoriale memorie-la-memorie cu operanzi maşină de lungime arbitrară (volum), cu una sau două dimensiuni. O singură instrucţiune conţine pînă la 4 operaţii diferite, şi pînă la 5 operanzi diferiţi. De exemplu, codul FORTRAN :

```
      DO 10 I=2, N-1
      DO 10 J=2, N-1
10    NEWB (I, J) = (B(I-1, J)+B(I, J-1)+(B(I, J+1)+B(I+1, J))/4.
```

se va compila într-o instrucţiune BSP de nivel înalt. Funcţie de N, o variabilă ce se determină în cursul execuţiei, instrucţiunea va genera o secvenţă de semnale de control pentru fiecare set de 16 elemente. Aceste semnale comandă memoria, reţelele de aliniere şi unităţile aritmetice. Mai mult, aceste semnale se activau în aşa fel încît să optimizeze suprapunerea în pipeline.

În plus faţă de operaţiile monadice, diadice, tetradice, pentadice asigurate de repertoriul de instrucţiuni BSP, existau de asemenea un număr de instrucţiuni vectoriale speciale. Cele mai importante sînt trecute în tab. 3.7.

În figura 3.37 se prezintă formatul instrucţiunilor BSP, inclusiv cîmpurile pentru lungimea buclelor, forma instrucţiunii şi operanzii folosiţi; descriptorii pentru fiecare operand şi rezultat. Opţional pot fi specificaţi şi vectori booleeni pentru operanzi şi rezultate. Vom folosi exemplul din fig. 3.37 pentru a ilustra modul de execuţie al acestei instrucţiuni care realizează o înmulţire matricială.

Primul lucru care trebuie notat este că instrucţiunile specifică 3 bucle DO încuibărite. Astfel, această instrucţiune de calcul va fi plasată în interiorul unei bucle în codul scalar care, cînd se va executa pe procesorul

Tabelul 3.7 Operaţiile vectoriale speciale de la BSP.

Operaţii scalare vectoriale	$A \leftarrow B$ op S
Reducere cu simplă precizie	$S \leftarrow A_1 + A_2 + \ldots A_n$
Reducere cu precizie dublă	Ca mai sus dar în dubla precizie
Secvenţa	$A_i \leftarrow A_1 + A_2 + \ldots + A_i$
Comprimare	$A \leftarrow B$ (sub control boolean)
Expandare	$A \leftarrow B$ (,,)
Combinare	$A \leftarrow B$ sau C (,)
Extragere aleatoare	$A \leftarrow B(I)$
Memorare aleatoare	$A(I) \rightarrow B$
Produsul scalar	$S \leftarrow A_* B_1 + \ldots + A_n * B_n$
Recurenţa (toate)	$A_1 \leftarrow C_1, A_i \leftarrow A_{i-1} * B_i + C_i$
Recurenţa (ultimă)	Ultimul termen din secvenţa de mai sus
Trasfer de daztă	Memoria de control $\Leftrightarrow$ memoria paralelă

scalar va genera 32 copii ale instrucţiunii cu cîmpurile de start parametrizate completate. Aceste cîmpuri vor fi create de procesorul scalar în timp ce instrucţiunea se află în buferul vectorial de 120 biţi. Odată asamblată, instrucţiunea este trecută unităţilor de control ale masivului pentru prelucrări ulterioare şi execuţie.

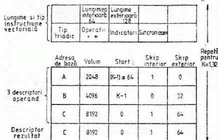


Fig. 3.37 Formatul limbajului maşină de nivel înalt BSP şi descriptorii vectoriali: (sus) cod FORTRAN; (jos) forma vectorială BSP.

Primul etaj al unităţilor pipe de control asamblează secvenţa de descriptori furnizaţi de unitatea de prelucrare scalară într-un descriptor unic, global, al operaţiilor. De asemenea stabileşte orice dependenţă între instrucţiunile vectoriale succesive. Cînd această etapă este încheiată rezultatul este plasat într-o coadă de aşteptare denumită memoria descriptorilor

de tipare, unde se așteaptă execuția. În această etapă descrierea este încă la nivelul programului. Acesta a fost translatat numai în operații cu mulțimi de 16 elemente numai în faza finală de prelucrare a instrucțiunii, de către unitatea de control a tiparelor (template control unit), care a selectat secvența de microinstrucțiuni corespunzătoare operației, pe care o ciclează apoi incrementînd și decrementînd adrese și numărătoare de bucle.

Pentru orice formă de instrucțiune există un număr de secvențe de microinstrucțiuni alternative, păstrate în ROM. Aceste tipare și procesorul de control al tiparelor selectează tiparul optim pentru fiecare etapă de execuție a instrucțiunii. Aceste semnale de control au fost denumite tipare deoarece definesc zone din unitatea pipeline a masivului în timp, care trebuie parcurs cît mai strîns pentru a maximiza utilizarea unității pipeline.

Fig. 3.38 prezintă 2 tipare ce pot fi folosite pentru execuția instrucțiunii din exemplul nostru (fig. 3.37). Ele definesc numărul ciclilor necesari pentru fiecare secțiune a unităților pipeline și modul de suprapunere. Deoarece unitatea forma o rețea inel și s-a folosit o singură memorie atît pentru extragerea cît și pentru memorare, trebuie să se facă rezervări, lucru arătat de liniile întrerupte. Aceste zone nu trebuie să se suprapună. Al doilea tipar din fig. 3.38, are un spațiu liber între extragerea lui A și a lui B; acesta este necesar tocmai pentru o rezervare de la execuția unui tipar anterior.

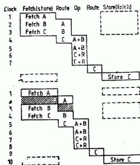


Fig. 3.38. Două tipare pentru calcularea expresiei $C = C + A * B$ la BSP. Se prezintă și perioadele de ceas necesare în unitatea pipeline BSP. Extragerea și memorarea se execută în aceeași memorie și nu pot fi suprapuse, de unde al doilea tipar care rezervă un ciclu de memorare pentru un tipar anterior.

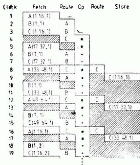


Fig. 3.39 O ilustrare a modului cum se pot suprapune tiparele. Operația care se execută este o înmulțire matricială (vezi de asemenea fig. 3.37 și 3.38).

Reîntorcîndu-ne la exemplul nostru, putem vedea cum unitatea de control a tiparelor selectează tiparul adecvat și incrementează adresele. Acest lucru îl arată fig. 3.39. Presupunem că primul tipar este lansat în

execuţie. Unitatea de control va selecta deci prima din cele două alternative din fig. 3.38, deoarece aceasta minimizează numărul de perioade de ceas necesare. Pentru extragerea următorului set de 16 operanzi nu este nevoie de nici o rezervare, de aceea se foloseşte acelaşi tipar a doua oară. Observăm că aici unitatea aritmetică este resursa critică. Toate tiparele succesive sînt de al doilea tip, care necesită rezervare pentru ciclul de memorare. Se poate vedea că după acest moment toate unităţile aritmetice şi de memorie sînt folosite.

Fig. 3.39 mai prezintă o restricţie de proiectare la suprapunerea operaţiilor de I/E cu cele de acces la memorie, deşi toţi ciclii memorie şi aritmetici sînt folosiţi, reţelele de transfer sînt ambele utilizate. Deşi pentru fiecare ciclu de memorie se foloseşte o perioadă de ceas, operaţiile de transfer nu pot fi suprapuse cu cele de acces la memorie cu un singur comutator. Aceasta deoarece operaţia de transfer are loc în cazul extragerii după cea de acces la memorie, iar în cazul memorării înaintea acesteia. Astfel vor fi necesare ambele reţele de comutare, care întotdeauna vor fi exploatate parţial şi nu la întreaga capacitate. În acest mod pot fi optimizaţi numai ciclii aritmetici şi de acces la memorie.

Pentru performanţa calculatorului BSP în cazul vectorilor scurţi este important ce se întîmplă între instrucţiuni. Să presupunem că înmulţirea matricii este urmată de buclă :

```
DO 20 I=1,200
20 D(I)=E(I)*F(I)
```

Unitatea de control a masivului BSP va determina dacă există dependenţe secvenţiale între instrucţiunile succesive şi dacă nu există posibilitatea unor hazarduri vectoriale va suprapune execuţia instrucţiunilor. Acest lucru este ilustrat pentru exemplul nostru în fig. 3.40. Se poate vedea că atunci cînd instrucţiunea este complet operaţională accesele la memorie sînt limitate, iar în cursul suprapunerii, folosind cele două tipare din fig. 3.41, s-au utilizat toţi ciclii de memorie. În mod similar dacă operaţia este dominată de calcule aritmetice, optimizarea va folosi toţi ciclii unităţilor aritmetice.

Concepţia de ansamblu a calculatorului BSP a fost de a se asigura în mod continuu o performanţă înaltă, fără întreruperi datorită operaţiilor de I/E sau sistemului. Astfel, interpretarea cifrelor de performanţă fără a ţine cont de această observaţie poate induce în eroare. Burroughs nu a urmărit atingerea unor performanţe foarte mari, care să nu poată fi menţinute continuu ci să proiecteze un calculator care să realizeze permanent 50—100% din performanţa sa maximă şi aceasta chiar programat exclusiv în FORTRAN.

Timpii elementari pentru unităţile aritmetice sînt furnizaţi în tab. 3.3.8. Trebuie să remarcăm că toate overhead-urile asociate acestor operaţii (de exemplu, accesul la memorie şi alinierea datelor) se suprapun în oarecare măsură. De aceea aceste cifre reprezintă performanţele ce pot fi atinse continuu. De exemplu, pentru +, — şi * BSP poate executa operaţiile triadice memorie-la-memorie cu viteza de 50 Mflop/s. Pentru operaţiile diadice cu accesuri limitate la memorie era posibilă o rată continuă de 32 Mflop/s.

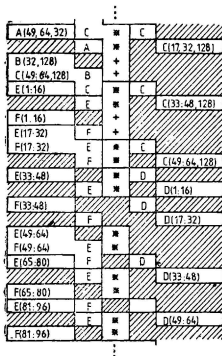


Fig. 3.40 Modul cum pot fi suprapuse diverse operații. Aici, operația triadică care continuă din fig. 3.39 se suprapune cu o operație diadică care folosește tiparul din fig. 3.41.

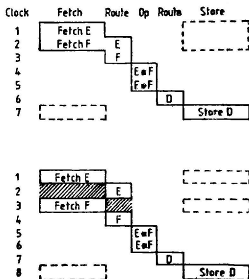


Fig. 3.41 Tipare pentru operația diadică **D** = = **E*F** la BSP.

Tabelul 3.8 **Operații la BSP.**

Operația	Timpul pentru 16 rezultate (ns)	Viteza de execuție pentru diadă (Mflop/s)	Viteza de execuție pentru triadă (Mflop/s)
+	320	33	50
—	320	33	50
—	320	33	50
÷	1280	12,5	12,5
5	2080	7,7	7,7

Deși overhead-urile sînt suprapuse într-un pipeline, operațiile vectorizate nu consumă deloc sau foarte puțin timp de inițializare. Motivul este că BSP era capabil să suprapună chiar și operații complet diferite. Fig. 3.42 și 3.43 ilustrează acest lucru prezentînd timpii reali pentru operațiile vectoriale de lungimi diferite. În fig. 3.42 unitatea pipeline a fost artificial încărcată înainte și după o singură operație vectorială. Se poate vedea că se obține un $n_{1/2}$ de 150, datorită timpului de inițializare asociat cu rețeaua

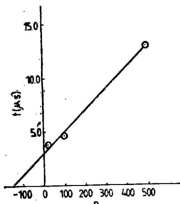


Fig. 3.42 Măsurători ale lui $n_{1/2}$ și r_{∞} pentru BSP, cu o singură instrucțiune vectorială și unitatea pipeline golită înainte și după operație; $n_{1/2} = 150$, $r_{\infty} = 50$ Mflop/s. (Date puse la dispoziție de Burroughs Corporation.)

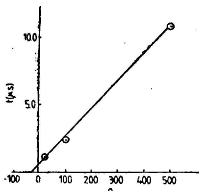


Fig. 3.43 Măsurători ale lui $n_{1/2}$ și r_{∞} la BSP, cu operații vectoriale succesive suprapuse; $n_{1/2} = 25$, $r_{\infty} = 48$ Mflop/s. (Date furnizate de Burroughs Corporation.)

pipeline în inel. Totuși, în utilizarea curentă, operațiile vectoriale care se succed sînt suprapuse folosind descriptori de tipare. Efectele sînt prezentate în fig. 3.44 unde se vede că $n_{1/2}$ s-a redus la 25.

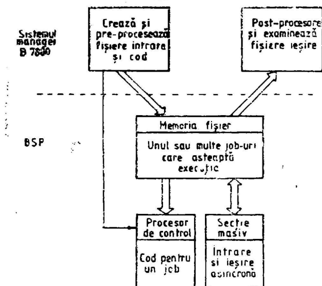


Fig. 3.44 Ciculația datelor și a controlului într-un program BSP tipic.

3.4.4. Denelcor HEP

Denelcor Heterogeneous Element Processor (HEP) a fost proiectat de Burton Smith (1978). Deși în 1985 proiectul a fost întrerupt datorită unor probleme financiare, arhitectura (ca și la BSP) este suficient de in-

teresantă pentru a asigura o tratare detailată în acest capitol. La nivelul cel mai general (fig. 3.45), HEP constă din 16 module de execuţie a procesełor (PEM) conectate printr-o reţea de comutare a pachetelor cu pînă la 128 module de memorie pentru date (DMM). Cele 16 PEM execută programe separate, dar toate pot accesa date din oricare din cele 128 DMM care formează împreună o memorie partajată de mare capacitate. Fiecare DMM conţine pînă la un milion cuvinte pe 64 biţi cu un timp de acces de 50 ns. Un PEM accesează memoria prin trimiterea unui pachet ,,cerere " pe 128 biţi care conţine adresa datei (de ex. adresa DMM şi adresa din interiorul DMM) Aceasta traversează reţeaua cu mai multe etaje pînă la DMM corect, data solicitată este accesată şi introdusă în pachet, care este returnat modulului PEM solicitant, tot prin reţea. Timpul de parcurgere al unui nod este de 50 ns, iar un interval de timp tipic care se scurge de la cererea datei pînă la introducerea ei în PEM este de 2,4 µs.

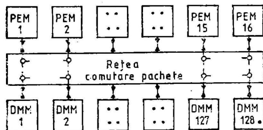


Fig. 3.45 ○ reprezentare bloc a sistemului Denelcor HEP, cu pînă la 16 PEM conectate de o reţea de comutare a pachetelor la pînă la 128 DMM.

La acest nivel de descriere HEP poate fi considerat ca un sistem MIMD cu memorie comună, şi cu reţea de comunicare cu mai multe etaje, conform descrierii din § 1.2.6.

În orice sistem MIMD, aspectele de proiectare cele mai importante se referă la mecanismele de sincronizare şi protocoalelor asociate pentru accesul multiplu la memoria comună. La Denelcor HEP, ambele mecanisme folosesc un indicator plin/gol asociat cu toate cuvintele din memorie. Acest dispozitiv permite implementarea protocolului handshake la fiecare cuvînt de date din întregul sistem; data nu poate fi citită dintr-o locaţie decît dacă acest bit semnifică ,,plin", iar scrierea nu poate avea loc decît dacă bitul semnifică ,,gol". În mod normal, după o operaţie de citire bitul este făcut ,,gol" iar după o operaţie de scriere ,,plin". Se poate vedea că acest mecanism este suficient pentru implementarea protocolului necesar pentru un canal Occam (vezi § 4.4.2). De aceea este suficient pentru implementarea tuturor primitivelor de sincronizare şi protejare a datelor.

Structura unui PEM este interesantă deoarece fiecare poate prelucra pînă la 50 fluxuri de instrucţiuni, de la maximum 7 task-uri utilizatori. Fluxurile multiple partajează un pipeline de execuţie a instrucţiunii cu 8 etaje. Fluxurile de instrucţiuni sau procesele sînt comutate la fiecare perioadă de ceas, astfel că distribuţia resurselor procesorului este extrem de echitabilă. Organizarea acestei unităţi pipeline este prezentată în fig. 3.46. Astfel un PEM singur este un exemplu de calculator MIMD pipeline (vezi fig. 1.8.). PEM este controlat cu o coadă de aşteptare care conţine *descriptorii proceselor* (1 pentru fiecare flux de instrucţiuni). Descriptorul conţine *cuvîntul de stare* al programului pentru fluxul de instrucţiuni sau procesul pe care îl reprezintă. Printre alte informaţii mai conţine numărătorul de program, pentru acel proces, actualizat la fiecare pas prin INC PSW.

Instrucțiunile propriu-zise se află într-o *memorie program* de 1 Mw, iar datele locale se află fie în 2048 registre pe 64 biți, fie în 4096 locații de memorie read-only, rezervate pentru constantele cele mai des folosite. Unități funcționale pipeline separate execută înmulțirea, adunarea și împărțirea în virgulă mobilă, operații cu întregi (IFU), creează operații (CFU)

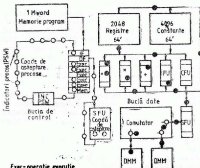


Fig. 3.46 Modul de funcționare al unui PEM de la HEP. Bucla de control apare în stînga, iar bucla de execuție în dreapta. Comutatorul asincron este decuplat de la operarea sincronă a unităților funcționale. (INC notează incrementul. IF notează extragerea instrucțiunii, iar DF notează extragerea datei. PSW notează cuvîntul de stare al procesului).

și execută referințe la memoria partajată (SFU). La parcurgerea descriptorilor de procese sînt apelate și datele necesare. Datele părăsesc registrele, parcurg unitatea pipeline corespunzătoare, iar rezultatul este returnat memoriei registru sub controlul descriptorului de proces interpretat de unitatea pipeline pentru instrucțiuni. Cînd descriptorul de proces trece prin primul etaj, instrucțiunea este extrasă din memoria program; în al doilea etaj se extrag datele adresate de instrucțiunea din registre sau memoria cu constante, care sînt transmise unității funcționale corespunzătoare; în timpul parcurgerii etajelor 3—7 datele sînt prelucrate și funcția îndeplinită; în sfîrșit, în etajul al 8-lea rezultatul este memorat înapoi în registre.

Descrierea anterioară se referă numai la instrucțiuni care accesează date din registre sau memoria pentru constante. Dacă o instrucțiune folosește date din memoria partajată (DMM) descriptorul procesului este scos din coada de așteptare a proceselor și introdus în coada de așteptare SFU, pînă la obținerea datelor de memorie. Se spune că descriptorul este îndepărtat (wave off) iar locul său din coada de așteptare a proceselor poate fi ocupat de alt proces, sau coada poate fi scurtată. Acest mecanism permite ca această coadă să conțină numai procesele active. În acest mod

fiecare cuvînt de stare al procesului (PSW) din descriptorul procesului poate păstra în permanență numărătorul de program actualizat la fiecare pas prin coada de control. În acest mod, toate operațiile sînt sincrone cu excepția accesului la memoria partajată. Aceasta este gestionată în coada SFU care reînserează procesul îndepărtat în coadă de așteptare, numai cînd pachetul cu date a sosit din DDM.

Un nou flux de instrucțiuni (denumit proces) se inițiază cu instrucțiunea mașină (sau FORTRAN) CREATE. Aceasta folosește CFU pentru a crea un descriptor de proces nou, pe care îl introduce în coada de așteptare a procesului. Se spune atunci că este un proces activ. Rămîne așa fie pînă cînd procesul este încheiat (de exemplu, o instrucțiune QUIT sau RETURN), fie descriptorul părăsește coada în timp ce așteaptă date din memoria partajată (wave-off). Spre deosebire de transputer nu există nici un mecanism pentru întîrzierea pasivă a unui proces, pînă la marcarea unui time-out.

Este instructiv să examinăm performanța unității pipeline pentru instrucțiuni odată cu creșterea numărului proceselor active. În cazul unui singur task (colecție de procese utilizator) coada de așteptare a proceselor poate fi considerată ca o coadă circulară cu pînă la 50 descriptori. Lungimea minimă a cozii de așteptare a proceselor este 8 și dacă sînt active mai puțin de 8 procese, în coadă vor exista locuri goale. Acestea vor fi ocupate prin crearea unor procese suplimentare (pînă ce vor fi ocupate 8 procese), fără a modifica lungimea cozii, sau timpii de execuție ai celorlalte procese. Astfel, cu cît se vor adăuga noi procese, viteza totală de prelucrare a instrucțiunilor va crește liniar cu numărul proceselor active, deoarece există mai puține locuri goale în coadă, și deci se vor prelucra în același timp mai multe instrucțiuni. Acest proces continuă pînă cînd vor fi 8 procese active și unitatea pipeline este plină. În această situație viteza de prelucrare este maximă. O instrucțiune părăsește unitatea pipeline la fiecare 100 ns, obținîndu-se o rată maximă de 10 Mips pe PEM, sau 160 Mips pentru un sistem complet cu 16 PEM. Dacă se creează mai mult de 8 procese active singurul efect este creșterea lungimii cozii de așteptare a proceselor. În acest mod viteza de prelucrare a instrucțiunilor scade, pentru fiecare procesor lăsînd viteza de execuție totală a instrucțiunilor constantă la 10 Mips pe PEM.

Pentru a rezuma acest mecanism, viteza de prelucrare va crește liniar cu numărul proceselor active pînă cînd unitatea pipeline este plină, după care rămîne constantă. Unitatea pipeline devine plină cînd există 8 procese active. Deoarece cele mai multe procese active vor folosi memoria partajată și deci vor fi, în anumite momente, îndepărtate din coadă, practic vor fi necesare mai mult de 8 procese active pentru a menține plină unitatea pipeline. De fapt, în programele FORTRAN sînt necesare 12—14 procese active, care ar permite îndepărtarea momentană a 4—6 procese.

Frumusețea acestui sistem este aceea că, chiar dacă un anumit utilizator nu poate furniza procese suficiente, pentru a umple unitatea pipeline, ea va fi încărcată în mod automat cu procese din alte task-uri sau job-uri Astfel, HEP, poate lucra în multi-tasking la nivelul instrucțiunii, într-un singur procesor, fără overhead. Bineînțeles va exista overhead pentru crearea și distrugerea proceselor.

Pentru a deduce performanța efectivă a calculatorului HEP în cazul aplicațiilor dominate de operații în virgulă mobilă, este necesar să știm cîte instrucțiuni sînt necesare pentru o operație în virgulă mobilă. Această variabilă, notată cu i_3, modifică astfel performanța asimptotică pe PEM :

$$r_\infty = 10/i_3 \text{ Mflop/s}$$

Evident i_3 trebuie să fie minimizat pentru a maximiza performanța în virgulă mobilă. Să considerăm cazul unei operații vectoriale diadice în care toate variabilele sînt memorate în memoria partajată. Aceasta poate fi exprimată cu bucla :

```
DO 10 I=1,N
10(A(I)=B(I)*C(I)
```

Deoarece HEP nu are instrucțiuni vectoriale, această buclă trebuie programată cu instrucțiuni scalare. Dacă bucla era programată în limbaj de asamblare, s-ar fi folosit 6 instrucțiuni :

(a) extrage B(I) din memoria partajată și depune în registre;
(b) extrage C(I) din memoria partajată și depune în registre;
(c) execută o înmulțire scalară registru la registru;
(d) memorează rezultatul, A(I), în memoria partajată;
(e) incrementează I;
(f) testează și efectuează salt la începutul buclei.

În acest caz $i_3 = 6$ și $r_\infty = 1{,}7$ Mflop/s. Dacă toate variabilele s-ar fi aflat în registre, instrucțiunile (a), (b) și (d) nu ar mai fi fost necesare, făcînd $i_3 = 3$ și dublînd performanța asimptotică la 3,3 Mflop/s. Optimizarea poate continua, obținîndu-se rate de prelucrare de 4,5 și 6,7 Mflop/s pentru operații cu memoria, respectiv registre, folosind 4 operații pe buclă. Sorensen (1984) și Dongarra și Sorensen (1985) au arătat că poate fi atinsă o performanță de între 5 și 6 Mflop/s pentru o serie de probleme cu matrici și o utilizare rațională a registrelor PEM, ca spațiu de memorie temporar pentru rezultatele vectoriale intermediare. Astfel, se reduce numărul referințelor la memoria partajată, ca și valoarea lui i_3.

Performanța se poate degrada cînd se folosesc mai multe procese separate pentru rezolvarea unei singure probleme, deoarece trebuie luat în considerare timpul necesar sincronizării fluxurilor separate de instrucțiuni, așa cum s-a arătat în § 1.3.6. S-au executat teste (r_∞, $s_{1/2}$) pentru a măsura overheadul de sincronizare la HEP. La acest test, se împarte activitatea caracteristică unei operații de înmulțire vectorială diadică memorie la memorie între P procese. Timpul de execuție corespunde expresiei :

$$t = r_\infty^{-1}(s + s_{1/2})$$

unde atît r_∞ cît și $s_{1/2}$ sînt funcții de P. Timpul măsurat include lansarea în execuție a P instrucțiuni la fiecare punct de sincronizare, ca și timpul necesar detectării încheierii activității de către toate fluxurile.

O analiză teoretică detailată a timpilor de execuție pentru testul anterior se găsește în lucrarea lui Hockney (1984a), iar valori măsurate pentru r_∞ și $s_{1/2}$ pentru o varietate de cazuri diferite în lucrarea lui Hockney și Snelling (1984) și mai pe larg în lucrarea Hockney (1985c). Rezultatele arată că pentru un număr fix de procese, t este o funcție liniară de s și

respectă modelul de mai sus. Pe de altă parte, fig. 3.47 prezintă variaţia timpului ca funcţie de numărul proceselor, pentru un s fixat. În acest caz, timpul descreşte întîi la un minim, odată cu completarea locurilor în pipeline. La $P = P_{opt} = 12$, timpul are valoarea minimă. Dacă ulterior P creşte, timpul creşte treptat datorită intervalului mai mare necesar sincronizării

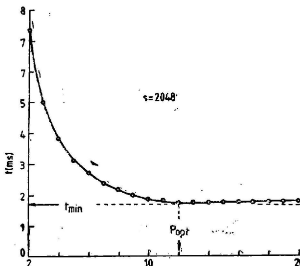


Fig. 3.47 Numărul optim de procese necesare pentru funcţionarea unui PEM.

unui număr mai mare de procese. Este clar, din acest exemplu, că nu se cîştigă nimic prin crearea a mai mult de 12–14 procese pe un singur PEM, chiar dacă logica programului ar indica un număr mai mare.

Fig. 3.48 arată valorile lui r_∞ şi $s_{1/2}$ ca o funcţie de P pentru sincronizare de bifurcaţie/joncţiune. La această metodă procesele se creează cu instrucţiunea FORTRAN CREATE, după care dispar odată cu încheierea calculelor. Performanţa asimptotică creşte liniar pînă la atingerea valorii maxime, după care rămîne constantă. Valoarea maximă $r_\infty = 1{,}7$ Mflop/s este consistentă cu $i_3 = 6$, aşa cum s-a menţionat anterior. Analiza teoretică (linia plină) anticipează că $s_{1/2}$ ar trebui să crească pătratic cu P, variaţie detectată şi în practică. Punctul de lucru cel mai bun, indicat de săgeată, se obţine la atingerea performanţei maxime. Orice creştere ulterioară a lui P determină sporirea overhead-ului (valoarea lui $s_{1/2}$), fără a îmbunătăţi valoarea lui r_∞. În punctul optim avem :

$$\text{FORK/JOIN } r_\infty = 1{,}7 \text{ Mflop/s şi } s_{1/2} = 828$$

Această metodă de sincronizare cere ca procesele să fie create şi distruse dinamic. Un proces este creat cu CREATE (FORK) şi lăsat să dispară la încheierea lui (JOIN). Evident această metodă de sincronizare este ineficientă, conducînd la valori mari pentru $s_{1/2}$. Alternativa este de a crea procesele numai în mod static şi de a realiza sincronizarea în alt mod.

Mecanismul de sincronizare este asigurat hardware, cu indicatorii plin/gol de la fiecare locație de memorie. Prin urmare, sincronizarea se poate realiza cu semafoare, prin utilizarea unor variabile comune. Un numărător inițializat cu numărul proceselor concurente poate fi decrementat cu unu la încheierea oricărui proces. Apoi toate procesele încheiate așteaptă ca variabila să devină zero, punct unde se realizează sincronizarea.

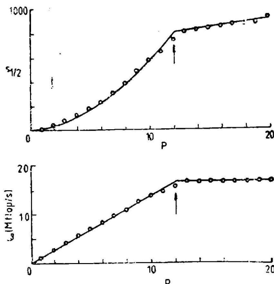


Fig. 3.48 Diagrame ce prezintă $s_{1/2}$ și r_∞ funcție de numărul de procese, P, în coada de așteptarea proceselor.

Aceasta este o implementare prin program a sincronizării, care are cu mult mai puțin overhead decît crearea și distrugerea dinamică a proceselor.

BARRIER $r_\infty = 1{,}7$ Mflop/s

și $s_{1/2} = 230$

Pentru a testa efectul creșterii numărului de operații aritmetice pentru o referință la memorie, s-a înlocuit insrucțiunea 10 din bucla (1.5) cu :

```
IF(A(I).LE.0.0) THEN
  C(I) = SIN(A(I)*EXP(B(I)))
ELSE
  C(I) = COS(A(I)*EXP(B(I)))
ENDIF
```

În acest caz toate operațiile aritmetice asociate cu evaluarea funcțiilor SIN, COS și EXP se execută între variabile temporare memorate în registre. Performanța efectivă în Mflop/s este de aproape trei ori mai mare :

FORK/JOIN $r_\infty = 4{,}8$ Mflop/s și $s_{1/2} = 710$

BARRIER $r_\infty = 4{,}8$ Mflop/s și $s_{1/2} = 190$

Este clar că în acest exemplu numărul mediu de instrucțiuni pentru o operație în virgulă mobilă s-a redus de la 6 la 2, în comparație cu varianta diadică vectorială.

3.5 Multiplicarea—viitorul oferit de tehnologia VLSI

3.5.1. Un start sub auspicii bune

Apariția circuitelor LSI și VLSI a dat un impuls important cercetării și dezvoltării arhitecturilor de masive de procesoare și multiprocesoare. În capitolul 1 s-au trecut în revistă proiectele de multiprocesoare, dispo-

nibilitatea microprocesoarelor ieftine și puternice avînd un efect catalizator asupra acestor cercetări. Totuși, impactul puternic al tehnologiei VLSI este anticipat în domeniul masivelor de procesoare. Cum s-a subliniat în capitolul 6, circuitele VLSI solicită o regularitate și repetitivitate mare, pentru a se putea reduce timpul de proiectare și a optimiza utilizarea suprafeței de siliciu. Masivele de procesoare seriale pe bit sînt candidați ideali pentru VLSI (și chiar pentru WSI), deoarece constau dintr-un număr mare de circuite relativ simple, multiplicate într-o modalitate foarte regulată, de obicei după o grilă bidimensională.

Confirmarea o oferă numărul în creștere de lucrări științifice ce descriu proiecte ce se bazează pe masive conectate după o grilă. De exemplu, University College din Londra a continuat să dezvolte proiecte pentru CLIP (Fountain și Goetcherian 1980, Fountain 1983), iar ICL continuă să dezvolte DAP, sistemul descris în §3.5.2. Alte proiecte includ GEC GRID (Robinson și Moore 1982, Arvind et al 1983). LIPP de la Universitatea Linkoping, Suedia (Ericsson și Danielson 1983)., NTT APP (Komdo et al 1983) în Japonia, connection machine de la Thinking Machines Corporation (care este descrisă în §3.5.3.) și proiectul RPA de la Universitatea Southampton (descris în §3.5.4) .În cadrul tuturor acestor proiecte s-au) dezvoltat sau se dezvoltă circuite VLSI, care conțin între 16 și 64 procesoare binare.

Se poate procura de pe piață o capsulă denumită GAP (NCR 1984) care conține 96 PE binare. Aceasta este un masiv 6×12 cu conexiuni între vecinii cei mai apropiați, care poate fi conectată în cascadă pentru a forma sisteme mai mari. Punctul slab îl reprezintă memoria intern mică, cîte 128 biți pentru fiecare PE, ca și lipsa pinilor pentru conectarea unei memorii RAM externe.

O altă realizare importantă, cu un impact major asupra sistemelor multiprocesor, o reprezintă transputerul INMOS (INMOS 1984). Acest circuit poate fi considerat fie ca un microprocesor pe 32 biți, ca element de bază pentru construcția sistemelor din noua generație, fie ca realizare hardware a unui proces OCCAM. OCCAM (INMOS 1984) este un limbaj de prelucrare paralelă ce se bazează pe CSP (Hoare 1984), descris într-o oarecare măsură în §4.4.2.

Transputerul INMOS se deosebește de circuitele similare prin aceea că a fost proiectat să exploateze posibilitățile oferite de tehnologia VLSI. Ca și alte microprocesoare produse recent are o arhitectură cu un set redus de instrucțiuni (RISC), iar spațiul economisit a fost folosit pentru crearea unei memorii RAM interne de capacitate mai mare, ca și (cu aceeași importanță) un sistem de comunicații pentru legături între transputere. Sistemul de comunicație este o implementare directă a protocolului asincron dintre procesele OCCAM. Prin urmare, procesele paralele OCCAM pot fi direct distribuite pe un set de transputere interconectate. Transputerul și sistemele înrudite sînt prezentate mai pe larg în §3.5.5.

Există deja mai multe proiecte care urmăresc să exploateze transputerul în cadrul unor sisteme paralele. Putem aminti cel puțin două proiecte Alvey mari, un proiect ESPRIT propus de Universitatea Southampton și multe proiecte finanțate de diverse firme. Unul din obiectivele urmărite de Jan Barron (unul din cofondatorii firmei INMOS) a fost ca transpute-

rul să se vîndă în cantităţi comparabile cu circuitele de memorie. Dacă se va realiza acest obiectiv, se va putea achiziţiona o putere de calcul mare (10 Mips) cu preţul unui circuit de memorie (aproximativ 10$). Transputerul T800, anunţat recent, execută 1—2 Mflop/s, iar dacă preţul lui va putea fi coborît la nivelul a 50$, nu se va putea subestima impactul lui asupra calculatoarelor de mare performanţă. Deja cinci producători europeni şi doi americani au pe piaţă produse care folosesc mai multe transputere, realizînd o putere de calcul apreciabilă, cu un preţ de cost acceptabil. Odată cu scăderea preţului transputerelor şi implicit şi al sistemelor, paralelismul sub forma sistemelor multiplicate se va răspîndi pe scară largă.

INMOS a planificat minuţios noile produse din gama transputerului şi, într-o mare măsură, sistemele construite pe baza transputerelor au un viitor asigurat. Pentru aceasta pledează legăturile de comunicaţie care lucrează la viteze standard (5, 10 şi 20 MHz), ceea ce asigură o interfaţă standardizată simplă între generaţiile de produse. Primul produs transputer, T414, şi succesorul lui, T800, sînt prezentate în § 3.5.5., iar limbajul OCCAM în §4.4.2. Cele două secţiuni trebuie citite împreună pentru a se înţelege conceptele ce stau la baza sistemelor cu transputere. De asemenea, în § 3.5.5., se abordează unele aspecte legate de construirea sistemelor cu transputere, cu exemplificare prin proiectele de la Universitatea Southampton. Trebuie să notăm că şi RPA, prezentat în § 3.5.4., foloseşte ca gazdă şi calculatoare de control pentru masiv transputere, exploatînd în acest mod avantajul paralelismului proceselor inerent în OCCAM, ca şi interfeţele standard pentru comunicaţii.

3.5.2. LSI DAP şi dincolo de el

În cursul anului 1985, Royal Signals and Radar Establishment (RSRE) din Malvern a achiziţionat primul sistem DAP 32×32, din a doua generaţie, construit pe baza tehnologiei LSI, pentru a-l folosi la aplicaţii radar şi alte prelucrări de semnale. În 1986 a fost instalată la DAP Support Unit de la QMC prima versiune comercială a aceleiaşi maşini. Acest prototip este interfaţat cu un sistem mono-utilizator ICL Perq, cu sistem de operare PNX, o implementare ICL a sistemului UNIX. Odată cu această livrare, ICL a continuat dezvoltarea tehnologiei DAP în cadrul unei firme noi, Active Memory Technology Ltd (AMT). Aceasta a reproiectat prototipul pentru scopuri comerciale. La sfîrşitul lui 1987 s-au livrat primele exemplare unor beneficiari. În timp ce prototipul mini-DAP (cum este cunoscut) conţine un circuit cu 16 PE, versiunea AMT foloseşte unul nou cu 64 PE.

Arhitectura sistemului mini-DAP este, la nivel de element procesor, în linii mari aceeaşi cu a sistemului DAP. Diferenţa principală constă în adăugarea unor căi de I/E rapide, ca cea întîlnită la GEC GRID (Robinson şi Moore 1982, Arvind et al 1983) şi Goodyear MPP (Batcher 1980). Se foloseşte un registru suplimentar, conectat între PE, pentru a forma un registru paralel cu deplasare, care traversează masivul de la sud la nord. Registrul este comandat separat de către elementele procesoare,

care primesc instrucţiuni executabile de la unitatea de control master (MCU), vezi § 3.4.2. Deplasările sînt comandate de unitatea rapidă de I/E care are şi rolul de memorie tampon pentru date şi de a reformata datele în cursul transferului („on fly"). Datele pot fi deplasate în plan cu rata de 1 cuvînt pe 32 biţi în interiorul sau exteriorul masivului la fiecare 100 ns, ceea ce înseamnă o lărgime de bandă de 40 MB/s. Cînd planul constituit din aceste registre a fost încărcat de unitatea de I/E, poate fi memorat de către memoria sistemului DAP prin furtul unui ciclu. Cu alte cuvinte, operaţiile de I/E rapide pot avea loc simultan cu cele de prelucrare, ceea ce permite folosirea sistemului pentru aplicaţii în timp real, ca prelucrarea imaginilor sau a semnalelor.

Acest mecanism de I/E reprezintă o modificare importantă a filozofiei sistemului DAP. Primele calculatoare DAP erau implementate ca o componentă a unui alt sistem, de exemplu ICL 2900, care îi asigură şi serviciile de I/E (vezi § 3.4.2). Pe de altă parte, mini-DAP este un procesor extensie, care poate fi interfaţat cu o varietate largă de sisteme gazdă, sau poate funcţiona de sine stătător, funcţie de aplicaţia pentru care este folosit. În fig. 3.49 se prezintă o diagramă bloc a sistemului mini-DAP, care evidenţiază căile importante de date. Calea rapidă de I/E lucrează ca un convertor serial/paralel. Cu excepţia acestei căi de date, masivul este acelaşi ca la DAP. Reţeaua de interconectare a procesoarelor este de tipul grilei, iar căi de date pe linie şi coloană conectează elementele procesoare la celule binare din MCU şi HCU (unitatea de conectare a calculatorului gazdă). Într-adevăr, mini-DAP este compatibil cu predecesorul său la nivelul codului sursă scris fie în FORTRAN (DAP), fie în limbaj de asamblare (APAL).

Prototipul mini-DAP este implementat în tehnologie TTL şi CMOS, masivul de elemente procesoare fiind constituit din submasive 4×4

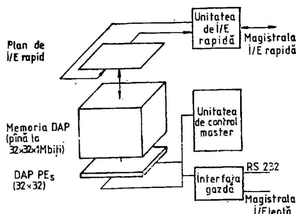


Fig. 3.49 Structura calculatorului ICL mini-DAP, cu un plan de I/E rapid şi un bufer.

implementate în circuitul National 6224. Acest circuit foloseşte tehnologie CMOS. Folosirea acestei componente, împreună cu progresele înregistrate de tehnologia memoriilor în ultimii 5–10 ani a determinat o reducere substanţială a numărului de capsule folosite de sistemul mini-DAP. Primul calculator DAP utiliza în medie 5 capsule pentru un PE, în timp ce

Fig. 3.50 Fotografia prototipului mini-DAP, care ilustrează aspectul fizic al cabinetului. (Fotografie pusă la dispoziţie de AMT Ltd.).

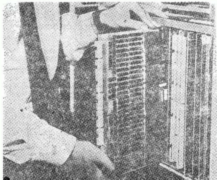


Fig. 3.51 Fotografie a unei plăci masiv de la mini-DAP, ce prezintă tehnologia folosită. Capsulele pătrate, mai mari, sînt masive de porţi LSI care conţin 16 procesoare. (Fotografie pusă la dispoziţie de AMT Ltd.)

mini-DAP foloseşte un masiv de porţi şi 8 circuite de memorie (opţiunea de 1MB) pentru un sub-masiv 4×4. Această reducere a ordinului de mărime, combinată cu dimensiunea mai mică a masivului au permis introducerea maşinii într-un cabinet de birou. Sistemul consumă mai puţin de 1 KW. Fig. 3.50 şi 3.51 prezintă aspectul fizic, respectiv tehnologia folosită.

La următorul nivel de implementare se află plăcile care conţin fiecare cîte 128 PE organizate ca un masiv 16×8. Placa mai conţine 128×8 K memorie necesară în cazul opţiunii de 1MB, sau 128×16 K pentru opţiunea de 2MB. Masivul 32×32 este format din 8 plăci. Restul sistemului este implementat pe încă 8 plăci; două sînt folosite pentru decodificarea instrucţiunilor, verificarea parităţii etc.; alte două formează MCU şi memoria program separată; cele rămase reprezintă interfeţele, HCU şi unitatea de I/E rapidă.

Principalele două diferenţe arhitecturale faţă de sistemul DAP se află la nivelul MCU şi al unităţilor de control suplimentar, cum sînt HCU şi unitatea de I/E. Aceste facilităţi fac ca mini-DAP să fie considerabil mai puternic decît DAP. De exemplu, la nivelul MCU există instrucţiuni suplimentare, ca cea de înmulţire, deplasare cu n poziţii şi pentru gestiunea întreruperilor. La DAP microcodul memorat în masivul DAP era extras

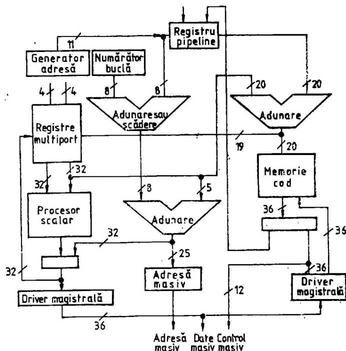


Fig. 3.52 Unitatea de control de la mini-DAP.

şi executat într-un ciclu cu două faze, dar la mini-DAP există o memorie separată pentru cod, care nu adaugă deloc overhead la ciclul instrucţiune. Memoria pentru cod este în mod obişnuit de 32 K×36 biţi, deşi există posibilitatea adresării unui spaţiu de 1 Mw. Noul MCU este ilustrat în fig. 3.52.

Fig. 3.53. prezintă unitatea rapidă de I/E de la mini-DAP, care se interfaţează cu magistralele de intrare şi ieşire pe 32 biţi, care formează planul de I/E. Conţine două bufere de 64 KB fiecare, care pot fi configurate separat sau împreună. Prin program ele pot realiza reformatarea datelor transmise masivului DAP. O astfel de posibilitate (întoarcerea la 90° — corner turning) este prezentată schematic în fig. 3.54.

Prototipul mini-DAP are ciclurile instrucţiune de 155 ns, cu aproximativ 25% mai rapid decît la DAP, deşi viteza maximă de prelucrare este mai mică pentru că şi numărul elementelor procesoare s-a redus la un sfert. Unele valori de performanţă sînt date în tab. 3.9.

Odată cu creşterea densităţii de integrare la noul sistem AMT DAP, va fi posibilă realizarea unei maşini cu aceleaşi dimensiuni, dar care să conţină 4096 PE, ca la DAP. Această maşină va avea şi un ceas mai rapid, cu perioada probabilă de 100 ns. Combinarea acestor elemente va produce o creştere de 6 ori a performanţelor faţă de cifrele furnizate. Totuşi, este improbabil ca aceste maşini să concureze pe piaţa ştiinţifică, unde sînt necesare operaţii în virgulă mobilă. T800 ar putea realiza performanţele enumerate anterior cu 5 pînă la 10 circuite. Aplicaţiile unde arhitecturile seriale pe bit sînt cele mai bune, sînt cele unde poate fi exploatată flexibilitatea lungimii cuvîntului; de exemplu, la prelucrarea semnalelor şi a imaginilor. Se poate vedea în tab. 3.9 că viteza de execuţie a adunării este invers proporţională cu lungimea cuvîntului.

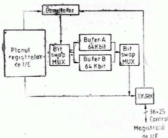


Fig. 3.53 Sistemul rapid de I/E al mini-DAP.

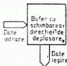


Fig. 3.54 Reprezentare a modului de reformatare, la 90°; buferul poate fi accesat după două direcţii ortogonale.

Alte aplicaţii unde excelează această clasă de arhitecturi sînt cele unde sînt necesare operaţii de emisie sau reducere. De exemplu, la prelucrările asociative, gestiunea bazelor de date etc.

Tabelul 3.9 **Cîteva cifre de performanţă pentru prototipul mini-DAP. Aceste valori sînt pentru un masiv 32 × 32 cu perioada ceasului de 155 ns. Toate cifrele sînt milioane de operaţii pe secundă.**

Lungimea Cuvîntului	Operaţie	Performanţa
8 biţi	Adunare	280
	Înmulţire	42
	Înmulţire (constanta)	100—200
16 biţi	Adunare	140
	Înmulţire	10
	Înmulţire (constanta)	30—100
32 biţi în virgulă mobilă	Adunare	8,6
	Înmulţire	5,1
	Ridicare la putere	10,9
	Rădăcină pătrată	7,4
	Împărţire	4,1
	Log	4,2

3.5.3. Connection machine

Connection machine a fost realizată pe baza studiilor efectuate la MIT de Hillis şi alţi cercetători, pentru o maşină paralelă dedicată aplicaţiilor de inteligenţă artificială. Deşi Hillis afirmă în lucrarea sa cu acelaşi titlu (Hillis 1985) că este „un nou tip de maşină de calcul", implementarea se bazează în mare măsură pe experienţa calculatoarelor seriale pe bit şi conexiuni după două direcţii ortogonale, care au precedat-o. Ceea ce distinge connection machine de predecesori este folosirea unei reţele de comutare complexe care asigură legături programabile între oricare două procesoare. Aceste conexiuni pot fi schimbate dinamic în cursul execuţiei programelor, ele folosind conceptul comunicaţiei prin pachete. Datele sînt transmise prin reţea, care are topologia unui hipercub cu 12 dimensiuni, către o adresă conţinută în pachet. Datorită topologiei reţelei, un bit al adresei binare corespunde la unul din cele două noduri situate după fiecare dimensiune a hipercubului (vezi §3.3.).

Connection machine îşi trage numele şi puterea din această abilitate de a stabili conexiuni arbitrare între procesoare; totuşi, s-au făcut multe compromisuri şi nu este clar dacă proiectanţii au suficientă experienţă în tehnologiile de implementare. Un hipercub cu 12 dimensiuni necesită o cantitate considerabilă de fire electrice şi, aşa cum s-a afirmat în capitolul 6, aceasta poate contribui la creşterea excesivă a costului şi degradarea performanţelor. Oricum, principiile care stau la baza maşinii sînt lăudabile, în asigurarea unei arhitecturi cu multiplicare virtuală, pe care poate fi distribuită în mod transparent descrierea problemei utilizator ce urmează a fi rezolvată.

Connection machine nu este singurul calculator serial pe bit la proiectarea căruia să se fi plecat de la problemele de reprezentare a informaţiei şi adaptabilitate. Proiectul RPA de la Universitatea Southamton (§3.5.4) este tot o arhitectură de masiv adaptiv care are un sistem de comunicaţii capabil să creeze conexiuni arbitrare între procesoare, şi să le modifice în

mod dinamic. Connection machine implementează conexiunile într-o rețea de transmisie a pachetelor serial pe bit, organizată într-o topologie lentă de cub binar, dar cu un grad mare de generalitate. Pe de altă parte, RPA implementează conexiunile prin comutarea circuitelor unei rețele conformă tehnologiei de implementare planară. Există și posibilitatea unor comunicații la distanță mare, care pot fi neregulate. Acestea sînt implementate cu prețul scăderii puterii de calcul, deoarece se folosesc elemente procesoare ca elemente de comutare a circuitelor. Acest ultim compromis nu trebuie să fie fixat în momentul proiectării sistemului. De exemplu, la connection machine, 50% din circuite sînt dedicate structurilor de comunicație. La RPA, întregul circuit este folosit pentru PE, dar în cursul compilării sau execuției, unele dintre ele vor fi folosite ca elemente de comutare.

Recomandăm cititorului să compare aceste două soluții ale problemei de comunicație. Conexiunea directă asigură o implementare hardware eficientă și realizează o lărgime de bandă mare. Pe de altă parte, comutarea pachetelor este mai costisitoare, dar asigură o utilizare mai eficientă a rețelei pentru o gămă largă de structuri de date. Mai mult, această soluție permite utilizarea implicită și dinamică a sistemului.

Connection machine a fost realizată de Thinking Machines Corporation, o companie co-fondată de Hillis. Prototipul, denumit CM—1, constă dintr-un masiv de celule procesor/memorie, care sînt conectate împreună cu o rețea de comunicații programabilă pentru a forma structuri dependente de date, denumite structuri active de date. Modul de lucru al acestor structuri este comandat de un calculator gazdă care accesează memoria masivului printr-o magistrală de date. Din acest punct de vedere, connection machine este similară sistemului ICL 2900/DAP și chiar sistemului transputer/RPA descris în continuare. În toate cele trei cazuri memoria masiv poate fi considerată ca aparținînd sistemului gazdă, astfel încît acesta să poată adresa și modifica locații ale memoriei masivului. Pentru task-uri cu un volum mare de calcule, gazda poate determina masivul să acceseze și prelucreze date aflate în această memorie partajată. În acest mod, se distribuie puterea de calcul în memoria sistemului gazdă, eliminîndu-se strangularea von Neumann care ar interveni altfel în cazul unei realizări uni—procesor cu viteze de calcul comparabile.

Masivul de memorie activă de la CM—1, DAP și RPA este o configurație foarte puternică, care permite execuția foarte rapidă a activității de sincronizare. Aplicații tipice ar impune emisia de informații în masiv, chei (de exemplu pentru activarea datelor), anumite prelucrări ale datelor active, urmate de o reducere a informației diseminate de sistemul gazdă, folosind selecția (adresarea memoriei masiv) sau reducerea (min, max, suma etc.).

Masivul procesor/memorie constă la CM—1 din 65 536 procesoare, fiecare cu 4096 biți de memorie. Aceasta este implementată cu circuite VLSI care conțin fiecare 16 PE și circuite RAM static de 4×4 K. În interiorul capsulei, elementele procesoare sînt conectate după direcțiile N, S, E și V cu o rețea denumită „NEWS", dar conform numărului de pini, specificat în lucrarea lui Hillis (1985), se pare că prototipul nu extinde această conectivitate și în exterior. Fig. 3.55 ilustrează această situație. Conexiunea hipercub binară asigură legătura la nivelul capsulei,

iar cele 16 PE din interior partajează lărgimea de bandă a unei scheme de transfer a pachetelor (router). De aceea, maşina poate fi considerată ca un hipercub cu 12 dimensiuni, cu noduri constituite din 4×4 PE binare. Fiecare capsulă conţine o schemă pentru transfer care gestionează mesaje de la cele 16 procesoare, către unul din cele 12 fire bidirecţionale. Prin urmare, reţeaua este formată din 4096 scheme de transfer, conectate prin 24 576 fire electrice. Aceasta înseamnă o densitate de fire foarte mare în comparaţie cu alte masive de procesoare seriale pe bit.

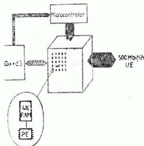


Fig. 3.55 Connection machine

Capsula procesor a fost implementată în tehnologie CMOS, are aproximativ 1 cm² şi conţine în jur de 50000 tranzistoare. Disipă 1W la 4MHz şi este introdusă într-o capsulă ceramică cu 68 pini; realizarea unei reţele NEWS izomorfice pe întreaga maşină ar necesita 32 pini suplimentari pe capsulă. Fiecare capsulă procesor este însoţită de 4 capsule de memorie statică $4\ K \times 4$, pe o placă aflându-se 32 capsule procesor şi 128 capsule de memorie, ceea ce înseamnă un total de 512 celule procesor/memorie. Modulele sînt plasate în sertare, cîte 16 în două sertare, conţinute de un rack. Maşina ocupă 4 rack-uri.

Ierarhia tehnologiei de implementare simplifică conectica pentru reţeaua cub, deoarece primele 5 dimensiuni se află pe o placă cu circuite, următoarele patru într-un sertar şi celelalte trei în rack-uri. Chiar aşa, la nivelul superior al cubului, fiecare muchie constă din 8192 perechi de fire, realizate cu cabluri bandă. Maşina este răcită cu aer, lucrează la 4 MHz şi disipă 12 KW. Comparînd frecvenţa ceasului şi dimensiunea masivului, ne-am putea aştepta la o performanţă cu un ordin de mărime mai mare faţă de cea de la DAP, dar pentru unele aplicaţii nici această performanţă nu a fost atinsă.

O unitate de control microprogramată conduce masivul, prin sincronizarea cu un ceas unic. Instrucţiunile de la gazdă (denumite macroinstrucţiuni) şi datele returnate sînt introduse într-o stivă FIFO, aflată între gazdă şi micro—controler. Aceste zone tampon echilibrează fluxul de microinstrucţiuni şi date dintre gazdă şi micro—controler. Fiecare apel la o micro—rutină, denumită de Hillis (1985) micro—instrucţiune poate avea nevoie de cîteva sau mai multe mii de perioade de ceas, funcţie de datele prelucrate.

Celula procesor de la CM—1, tipică multor sisteme seriale pe bit, este reprezentată în fig. 3.56. Operaţia de bază a celulei este o operaţie cu 5 adrese, cu biţii celor două surse şi ai destinaţiei furnizaţi de la, sau la memoria externă. Aceştia sînt adresaţi cu două cîmpuri a 12 biţi,

destinaţia avînd aceeaşi adresă cu unui din operanzi. Trei adrese pe 4 biţi sînt folosite pentru registre de intrare/ieşire ale ALU şi un registru indicator de condiţii; aceste adrese sînt interne capsului PE. Ultima adresă pe 4 biţi specifică care din registrele indicatoare de condiţii va fi folosit ca indicator condiţional sau de activitate. Această operaţie consumă trei perioade de ceas. Două cîmpuri de control pe 8 biţi definesc una din 256 funcţii booleene posibile, cu trei intrări pentru fiecare ieşire. Oricare din cele 16 indicatoare de condiţii poate asigura un control local on/off, iar un sumator realizează funcţia logică SAU a unui semnal de la toate procesoarele din masiv.

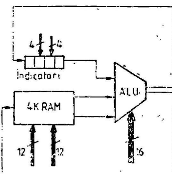


Fig. 3.56 Elementul procesor de la connection machine. ALU produce două ieşiri, care pot fi una din cele 256 funcţii boolene cu trei intrări.

Un minus al acestui calculator este exploatarea slabă a masei de fire care constituie reţeaua hipercub. Se poate afirma că firele de conectare reprezintă componenta cea mai costisitoare în sistemele moderne ce folosesc tehnologie VLSI. Datele sînt transferate ciclic, în cursul fiecărui ciclu trimiţîndu-se un mesaj după o direcţie. Aceasta înseamnă că se folosesc numai 1/12din totalul de fire. Evident, o schemă care ar transfera date simultan după cele 12 direcţii ar fi costisitoare, ceea ce face ca autorii să se îndoiască de viabilitatea unei reţele cu 12 dimensiuni. Cu 11/12 din lărgimea de bandă nefolosită, se poate obţine aproximativ aceeaşi lărgime de bandă pentru comunicaţii la distanţă, ca şi o lărgime de bandă pentru transmisii pe distanţe mici mai mare, cu o reţea NEWS de transmisie a pachetelor, cu transmisie şi recepţie concurentă după 4 direcţii. O astfel de reţea a fost implementată prin program, relativ eficient, pe RPA, sistem descris în § 3.5.4.

Tabelul 3.10 **Performanţa prototipului CM-1 cu 64 K procesoare.**

Caracteristica de performanţă	Valoare
Dimensiunea memoriei	$2{,}5 \times 10^8$ biţi
Lărgimea de bandă la memorie	$2{,}0 \times 10^{11}$ b/s
Lărgimea de bandă a procesorului *	$3{,}8 \times 10^{11}$ b/s
Lărgimea de bandă a comunicaţiei	
cazul cel mai defavorabil	$3{,}2 \times 10^7$ b/s
caz tipic	$1{,}0 \times 10^9$ b/s
cazul cel mai favorabil	$5{,}0 \times 10^{10}$ b/s

* Lărgimea de bandă a procesorului măsoară numărul de biţi care intră şi părăsesc ALU într-o secundă şi nu numărul de operanzi pe secundă.

Cititorul interesat poate găsi informaţii suplimentare privind Connection Machine în cartea lui Hillis (1985). În particular, funcţionarea reţelei de comunicaţii este descrisă mai pe larg. Tab. 3.10 prezintă performanţe ale sistemului CM—1, aşa cum le-a prezentat Hillis.

Mediul de programare se bazează pe LISP, limbaj cu o istorie veche la M.I.T. Connection Machine LISP (CMLISP) este o extensie a limbajului LISP, proiectată special pentru modul de lucru paralel al calculatorului. Este un limbaj SIMD care reflectă modul de funcționare a calculatorului gazdă și micro—controlerului, ce permite exprimarea operațiilor cu structuri de date paralele. CMLISP este față de LISP ceea ce este FORTRAN × (vezi § 4.3.3) față de FORTRAN 77. Limbajul este prezentat complet în lucrarea lui Hillis și Steele (1985), fiind prezentat complet în lucrarea prezentă în § 4.3.4.

3.5.4 RPA

(i) Masive de procesoare reconfigurabile

RPA este o arhitectură proiectată pentru prelucrarea structurilor definite în § 4.3; masivul folosește multe procesoare simple și își poate adapta structura fizică într-o oarecare măsură corespunzător structurilor de date. RPA este proiectat să prelucreze structuri de date, consistente cu rețeaua adaptabilă folosită; totuși restricțiile nu sînt chiar atît de severe pe cît ne-am aștepta, pentru că include astfel de structuri ca arborii binari. RPA a fost dezvoltat la Universitatea Southampton cu fonduri ale programului Alvey (Jesshope 1985, Jesshope 1986 a, b, c, Jesshope et al 1986, Rushton și Jesshope 1986, Jesshope și Stewart 1986, Jesshope 1987a, b). Este similar sistemelor DAP și connection machine în aceea că este un masiv de elemente procesoare binare cu un micro—controler comun. Nu este o mașină SIMD în totalitate, deoarece este posibilă modificarea locală a unei instrucțiuni prin distribuirea unor cîmpuri ale cuvintelor de control în masiv. Aceasta permite adaptarea masivului la situații diferite. Fig. 3.57 ilustrează acest concept pentru masive bi-dimensionale.

Este bine cunoscut că o structură sincronă ca aceea a unei mașini SIMD are dimensiunea limitată, deoarece impulsurile ceasului și informația de control trebuie distribuite pentru sincronizarea sistemului. Orice diferență la distribuirea acestor semnale va determina reducerea frecvenței ceasului. Există, prin urmare, o dimensiune optimă pentru RPA, care va depinde de caracteristicele de implementare și încapsulare. În mod obișnuit, masivele conțin pînă la 256 ×256 procesoare, cîte 16 pe o capsulă VLSI. Oricum, dimensiunea unui sistem RPA nu este limitată de dimensiunea blocului sincron optim, deoarece poate fi extinsă în continuare prin folosirea paralelismului procesului (vezi § 4.4).

RPA se află în interiorul sistemului de memorie al unui transputer, ca memorie activă, iar execuția proceselor pe RPA poate fi considerată ca o extensie a setului de instrucțiuni al transputerului în cadrul modelului de programare OCCAM. Un alt mod de exprimare ar fi că paralelizarea proceselor de către transputer și OCCAM este folosită pentru unirea mai multor sisteme sincrone capabile de o execuție eficientă a structurilor paralele. Prin urmare, RPA poate fi considerat fie ca un masiv de procesoare cu un transputer ca gazdă, fie ca un sistem de memorie inteligentă a transputerului, care poate fi multiplicată ca orice alt

sistem cu transputere. Multiplicarea poate exploata paralelismul explicit al OCCAM-ului, în ceea ce privește descrierea structurilor prin implementarea comunicațiilor prin legăturile INMOS. Tehnica paralelismului algoritmic (vezi § 4.5.2) poate fi exploatată în cadrul unui sistem cu mai multe RPA.

În orice sistem care folosește multiplicarea, problema conceptuală majoră este cea a repartizării procesoarelor virtuale definite în cadrul descrierii problemei pe o structură fixă. În acest sens, un aspect important al sistemului RPA este faptul că permite adaptarea structurii masivului la problema implementată. În acest mod se simplifică mult sarcina compilatorului, de distribuire a datelor pentru a obține identitatea între imaginea utilizatorului despre universul problemei, cu cea a mașinii. În același timp se realizează și o utilizare înaltă a hard-ului.

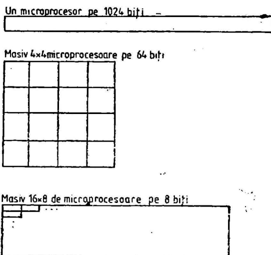


Fig. 3.57 Ilustrare a abilității RPA de a se adapta la diferite structuri dreptunghiulare.

În mod ideal, într-un masiv de procesoare pentru prelucrarea structurilor, ar trebui să fie un element procesor pentru fiecare dată a structurii prelucrate. Din păcate, nu există nici o posibilitate de a ști înainte cît de mare sau mică va fi structura. Ea poate varia dinamic. Structurile mai mari decît masivul sînt convenabile, deoarece fiecare proces poate prelucra secvențial atît de multe elemente ale structurii de date cîte sînt necesare. Intervin, totuși, probleme de menținere a topologiei structurii de date, în special dacă structura este complexă. Corolarul este abilitatea de a repartiza mai mult de un procesor unui element aparținînd structurii de date și acest lucru este mai puțin înțeles. De fapt, aceasta este diferența importantă dintre RPA și alte masive de procesoare seriale pe bit.

Pentru atingerea acestei flexibilități, s-a extins modelul de execuție SIMD prin aceea că procesooarele pot modifica instrucțiunea emisă. Procesoarele pot lucra într-un mod diferit, dar pre—programat local. Extinderea controlului local asigură atît flexibilitate cît și eficiență în cazul celor mai obișnuite operații de calcul și de comunicații. Se poate arăta că în afara faptului că structura masivului poate fi variată în formă și dimensiune, permite o implementare eficientă a operațiilor de manipulare a datelor, inclusiv cele cu structuri de date neregulate. De exemplu, alte topologii, ca arbori sau grafuri, pot fi implementate eficient, deși nu cu flexibilitatea dinamică a sistemului de comunicații cu pachete. Totuși, mecanismele pot fi exploatate pentru realizarea unor implemen-

țări software eficiente a comunicației cu pachete la RPA, așa cum se va arăta ulterior.

Soluția RPA la paralelismul cu structură virtuală permite execuția unui cod unic pe diferitele variante fizice ale sistemului, fără nici o modificare; mai mult, acest lucru este adevărat la nivelul inferior, deoarece rutinele microprogramate se pot adapta la configurații diferite ale dimensiunii și structurii masivului prin parametrizare.

(ii) *Implementarea RPA*

Masivul RPA este construit cu un element procesor foarte flexibil, sub forma unei latice bi-dimensionale, 4—conexă. În fig. 3.58 este prezentat un sistem RPA. Masivul a fost proiectat să lucreze ca un lanț de procesoare virtuale, conectate conform structurii ce este prelucrată. Fiecare PE trebuie considerat ca o secțiune binară a unei unități de prelucrare mai mare, care va executa o gamă largă de operații comune microprocesoarelor. Unitatea procesoare compusă execută operații binare, operații de deplasare și de calcul aritmetic pe bit. Aceste unități procesoare pot fi configurate dinamic, dacă este necesar, folosind orice cale care conectează elementele procesoare. Configurațiile cele mai utile sînt sub—grafuri apropiate structural masivului RPA, deoarece acestea permit execuția operațiilor de deplasare ciclică și de lungime dublă într-un micro—ciclu și simplifică semnificativ operațiile aritmetice seriale executate cu date memorate în unitățile procesoare. De exemplu, execuția serială a operațiilor aritmetice pe 16 biți cu unități procesoare pe 4 biți face ca bitul de transport de a bitul de rang trei al unei tetrade să fie adiacent bitului de rang 0, unde este necesar la următorul ciclu. Dacă se configurează procesoare pe mai mulți biți ca bucle închise de PE atunci configurațiile vor executa următoarele operații:

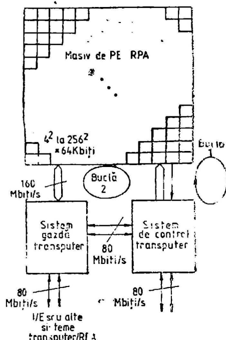


Fig. 3.58 Sistemul de calcul RPA, cu interacțiunile dintre masiv, controler și sistemul gazdă.

(1) toate operațiile logice binare cu doi operanzi (două operații programate independent pot fi executate într-un singur ciclu);

(2) deplasări ciclice și planare (aritmetice);

(3) deplasări ciclice și planare (aritmetice) de lungime dublă;

(4) copierea unui bit de către toți ceilalți în cadrul unui procesor; (5) operații aritmetice cu transport anticipat, prin tehnica lanțului de propagare rapidă a transportului Manchester, realizată prin operații mul-

tiple pe „bait", cu propagarea automată a transportului între sub—operații;
(6) înmulțire cu rezultat pe lungime dublă, prin folosirea adunărilor multiple cu transport anticipat (5), deplasări pe lungime dublă (3) și copierea bitului (4).

Elementul procesor are o structură cu 4 magistrale, așa cum se arată în fig. 3.59, două magistrale pentru operanzi și două pentru rezultate. ALU execută operațiile logice și aritmetice- folosind numai operanzi locali, iar mecanismul de selecție al vecinului permite transferul datelor de la un PE la altul. Operațiile aritmetice paralele pe bit sînt posibile prin folosirea atît a unității aritmetico—logice, cît și a selecției vecinului care, împreună, formează un sumator rapid Manchester. Cheia flexibilității masivului constă în folosirea cîmpurilor prestabilite de control, asigurate fiecărui PE din masiv, R, în fig. 3.59. Acestea sînt folosite la programarea comutatoarelor și la operațiile aritmetice și de deplasare.

Două din aceste cîmpuri, fiecare de doi biți, specifică modul de control al comutatorului vecinului cel mai apropiat (deplasări la stînga și dreapta); registrul de reconfigurare conține de asemenea un cîmp cu biți care definește semnificația elementului procesor în cadrul unității mai mari de prelucrare. Codurile folosite sînt pentru bitul cel mai semnificativ, cel mai puțin semnificativ și oricare alt bit. Codul rămas este folosit pentru o posibilitate foarte puternică care permite intrărilor comutatorului să fie conectate direct la ieșire. Astfel, se distribuie controlul comunicației, similar căilor de linie și coloană întîlnite la ICL DAP și GEC GRID. Totuși, acest lucru se realizează într-o manieră mult mai flexibilă, deoarece se poate folosi combinația codurilor de direcție și semnificație pentru interconectarea oricărui șir de PE. Conexiunile pot fi folosite fie pentru distribuirea datelor tuturor elementelor procesoare interconectate, fie pentru ocolirea PE pentru a implementa structuri de conectare care altfel nu ar fi posibile.

Memoria fiecărui PE funcționează ca o stivă. Există o stivă pe un bit și o stivă pentru activități, pentru memorarea datelor formate din-

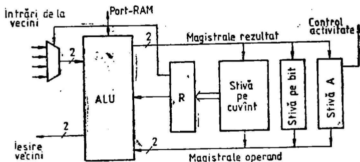


Fig. 3.59 Elementul procesor RPA (bit-slice).

tr-un bit, în fiecare PE. Fiecare constă din 8 biți identici pentru date, cu excepția faptului că bitul din vîrful stivei de activitate poate fi folosit pentru decuplarea condiționată a elementului procesor. In PE mai există memorie și pentru date. Acestea formează o stivă pentru cuvinte pe 8

biţi care pot fi convertite paralel—serial şi serial—paralel cu o pereche de registre cu deplasare. Se pot executa operaţii aritmetice serial pe bit (sau serial pe cuvînt), fără a se înregistra incomoda inversare a biţilor înregistrată la folosirea stivelor. În sfîrşit, un port de I/E pe un bit permite conectarea a pînă la 64 K biţi de memorie externă la fiecare PE.

Memoria internă RAM are mecanisme de control dual care permit accesul aleatoriu sau în mod stivă la baiţii unui cuvînt prin control local sau global. De exemplu, stiva poate fi scrisă sau citită condiţional, funcţie de valoarea stivei de activitate. Structura de memorare mai conţine un registru cu deplasare cu n poziţii, comandat local şi circuite de comparaţie între cele două registre cu deplasare care realizează conversia serial—paralel. Aceste posibilităţi permit execuţia mult mai bună a operaţiilor în virgulă mobilă. Pentru o anumită durată a tactului şi o dimensiune a masivului, performanţa în virgulă mobilă poate fi la RPA de 10 ori mai mare decît la ICL DAP. Preţul plătit este complexitatea mai mare a elementului procesor. S-a fabricat, pînă în momentul redactării lucrării de faţă, o capsulă cu un singur PE. S-a folosit tehnologie CMOS 3μm, iar suprafaţa utilă este de 4 mm². Capsula cu 16 PE va ocupa aproximativ 0,8 cm². Fig. 3.60 prezintă acest circuit de test.

(iii) *Sistemul de control la RPA*

În general, masive de procesoare ca DAP sau Connection machine folosesc un singur controler. Astfel, se obţine o maşină foarte eficientă prin partajarea unui mecanism de control complex între multe procesoare. Totuşi, RPA diferă de această structură de control în două moduri.

RPA poate fi considerat la un nivel ca un sistem MIMD, cu fiecare „procesor" conţinînd un masiv de procesoare sub control SIMD. Această structură (MIMSIMD) a mai fost propusă şi de alţi proiectanţi (Lea 1986) şi permite exploatarea în cadrul aplicaţiilor a tuturor formelor de paralelism. Componenta de sincronizare poate exploata paralelismul structurii, ca masive, şiruri, liste şi arbori, iar nivelul MIMD sincronizat prin evenimente poate exploata paralelismul neregulat sau independent caracteristic multor algoritmi.

Deşi un sistem unic RPA transputer poate fi considerat ca un masiv SIMD convenţional, fiecare PE poate, conform unor condiţii determinate local, memora sau genera o parte sau tot cuvintul de control necesar operaţiilor ulterioare. Această modificare adaptivă a modului de control SIMD permite fiecărui PE să exercite o oarecare autonomie de acţiune. Cele mai multe calculatoare SIMD asigură la nivelul procesorului un control minimal, redus de obicei la un comutator dependent de date on/off. Această structură menţine avantajul sincronizării de granulaţie fină, în timp ce asigură o adaptabilitate limitată la diverse cerinţe de prelucrare în masiv.

Fig. 3.58 prezintă structura de control a unui singur RPA. Se pot identifica două bucle de control majore. Prima se află între masiv şi microsecvenţiatorul propriu. Ea este cea convenţională ; controlerul furnizează cuvinte de microcontrol şi adrese în masiv (o cale largă pentru date) şi recepţionează din masiv şi posibil de la alte dispozitive ale sistemului

semnale de condiţii. A doua buclă de control se referă la datele din memoria masivului, care poate fi accesată de procese în curs de execuţie pe sistemul gazdă. Această buclă permite o sincronizare de granularitate mai mare care poate implica evenimente de la alte sisteme transputer /RPA. Folosirea acestei bucle implică iniţierea unor acţiuni în masiv de către gazdă, care

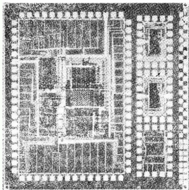


Fig. 3.60 Fotografie a circuitului de test RPA cu un PE şi structurile de test suplimentare. Capsula a fost fabricată la Universitatea Southampton.

vor iniţializa datele în memoria masivului ; ulterior aceste date vor fi regăsite de către masiv şi folosite pentru determinarea acţiunilor ulterioare.

Unul din obiectivele principale ale sistemului RPA a fost să utilizeze această a doua buclă de control în cadrul unui model de programare OCCAM, într-un astfel de mod încît programatorul să poată folosi concurenţa reală sau virtuală dintr-o aplicaţie. Astfel, codul pentru aplicaţii este portabil pentru diferitele configuraţii transputer/RPA.

Alternativele de implementare pentru structura de control RPA în OCCAM sînt determinate de granularitatea proceselor ce se execută în masiv. Trebuie să fie procesele programe complete, ce se execută concurent şi comunică probabil cu sistemul gazdă? Sau trebuie să fie considerate ca extensii indivizibile ale setului de instrucţiuni al sistemului gazdă? Discriminarea între aceste două alternative o asigură comunicaţiile necesare între gazdă şi masiv. Dacă definim un proces de bază în masiv ca o unitate indivizibilă de calcul, atît timp cît este implicată comunicaţia dintre

gazdă şi masiv, astfel că ordinea poate fi modelată de următorul fragment OCCAM

```
– HOST PERCEPTION OF ARRAY ORDER
ARRAY !SOMETHING

    ARRAY. PROCESS

    ARRAY ? SOMETHING. ELSE
```

atunci orice decizie referitoare la împărţirea controlului între gazdă şi controlerul masivului este de fapt o decizie dacă construcţiile ce definesc procesele de bază din masiv pot fi determinate de controlerul masivului sau de către transputerul gazdă.

Dacă controlerul masivului defineşte secvenţele formate din aceste procese, mergînd pînă la programe complete (procese) ce se execută pe masiv, atunci pentru operarea concurentă a sistemului gazdă şi masivului trebuie să existe comunicaţii între aceste două componente pentru asigurarea evenimentelor necesare sincronizării. În OCCAM se poate defini acest mod de lucru concurent :

```
PAR
  SEQ– –HOSTPROCESS
    ARRAY !SOMETHING
    HOST, PROCESS.1
    ARRAY ?SOMETHING.ELSE
    HOST.PROCESS.2

  SEQ– –ARRAY PROCESS
    ARRAY ?SOMETHING
    ARRAY.PROCESS.1
    ARRAY !SOMETHING.ELSE
    ARRAY.PROCESS.2
```

În acest exemplu sistemul gazdă iniţiază operarea masivului şi, apoi, îşi continuă lucrul. La un moment ulterior de timp cele două procese se sincronizează printr-o operaţie de comunicaţie iniţiată de masiv şi, apoi, ambele îşi continuă lucrul, probabil modificînd acţiunile pe care le execută în urma operaţiei de comunicaţie întreprinsă. Acest mod de control acceptă operaţii de comunicaţie pentru sincronizare iniţiate fie de gazdă, fie de masiv.

Dacă dorim suprapunerea calculelor cu operaţia de comunicaţie, astfel ca sincronizarea să nu scadă eficienţa, atunci modelul trebuie modificat pentru a permite atît gazdei cît şi masivului să execute în paralel procesele respective. Ilustrarea o oferă exemplul de mai jos, unde operaţiile de comunicaţie pentru sincronizare nu sînt vizibile :

```
PAR
  PAR– –HOST PROCESS
    HOST, PROCESS.1
    HOST.PROCESS.2
  PAR– –ARRAY PROCESS
    ARRAY.PROCESS.1
    ARRAY.PROCESS.2
```

Implementarea acestui model presupune un secvenţiator pentru masiv care să poată controla procese multiple, concurente. Acesta ar trebui să fie întreruptibil, astfel ca un proces care solicită efectuarea unei operaţii de comunicaţie cu sistemul gazdă să poată fi suspendat în timp ce are loc această operaţie şi să continue după terminarea ei. Overhead-urile asociate comutării proceselor sînt mari, iar microcontrolerul necesar complex. De aceea, am optat la RPA pentru alternativa ordonării indivizibile.

Această soluţie exclude sincronizarea între procesele în execuţie pe sistemul gazdă şi masiv, exceptînd iniţierea şi terminarea proceselor executate de masiv. Există un număr de avantaje. Deoarece în interiorul unui proces executat pe masiv nu este necesară sincronizarea, nici ordinea nu trebuie suspendată. Microrutinele se pot executa pînă la încheiere prin folosirea unui microsecvenţiator simplu. Este dezirabil un mecanism care să permită execuţia pe masiv a proceselor concurente, deoarece atît înainte de, cît şi după microrutină poate exista comunicaţie, care este de dorit să se suprapună cu operaţiile de calcul. Acest paralelism poate fi, totuşi, implementat cu o coadă de aşteptare pentru procese active; un secvenţiator întreruptibil va avea nevoie de cel puţin două cozi de aşteptare, pentru procesele active şi cele suspendate, cu mecanismul aferent pentru suspendarea şi activarea proceselor necesare sincronizării.

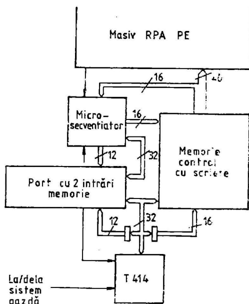


Fig. 3.61. Structura unităţii de control de la RPA.

Fig. 3.61 furnizează o imagine mai detaliată a sistemului de control de la transputer, care conţine o singură coadă de aşteptare pentru procese multiport, prin intermediul căreia comunică transputerul şi microcontrolerul. În cadrul acestei implementări, rutinele microprogramate de pe masiv formează procese care pot fi considerate extensii indivizibile la setul de instrucţiuni al calculatorului gazdă, şi care corespund limbajului OCCAM. Constructorii PAR şi SEQ, procesele de pe masiv şi gazdă pot fi acum amestecate cu operaţiile de comunicaţie (care nu apar în exemplul de mai jos) asigurînd evenimente de sincronizare şi operaţiile de control, cînd este necesar.

```
--HOST/ARRAY PROCESS
SEQ
  .
    .
      .
      PAR
        HOST.PROCESS.1
        SEQ
          ARRAY.PROCESS.1
          ARRAY?SOMETHING
          PAR
            HOST.PROCESS.2
            IF
              SOMETHING=GOOD
                ARRAY.PROCESS.2
              TRUE
                ARRAY.PROCESS.2
```

Cu acest sistem, o secvenţă de procese ce se execută pe masiv şi care nu au nevoie de operaţii de comunicaţie pentru sincronizare pot fi buferizate în controlerul masivului, controlul trecînd de la unul la următorul fără interacţiune cu sistemul gazdă. Ele sînt adăugate cozii de aşteptare a proceselor din controlerul masivului şi apoi executate în secvenţă. În acelaşi mod se pot executa şi procesele paralele.

La proiectarea sistemului de calcul RPA s-a implementat la nivelul inferior un volum considerabil de soft, dezvoltat prin folosirea unui sistem ce simulează RPA (Jesshope şi Stewart 1986). Se simulează un masiv 32×32 şi un controler, implementat în imaginea binară a microcodului. Se foloseşte echipamentul grafic de la o staţie de lucru ICL Perq. În fig. 3.62 este ilustrat menu-ul folosit. Interactiv, se furnizează date, se stabileşte configuraţia, se editează microcodul şi se simulează complet masivul şi controlerul, cu vizualizarea grafică a tuturor stărilor interne şi a magistralelor, cînd este necesar pentru depanare.

RPA ca staţie de lucru ce foloseşte transputere cu un masiv 32×32, va asigura între 20 şi 100 Mflop/s pentru operaţii cu numere pe 32 biţi în virgulă mobilă, la o frecvenţă a ceasului de 10 MHz. Deoarece operaţiile aritmetice se execută serial, performanţa maximă va depinde de precizia datelor. De exemplu, adunarea a 32×32 întregi pe 32 biţi se va executa cu viteza de 200 Mflop/s, în timp ce adunarea întregilor pe 8 biţi se va executa cu o viteză de patru ori mai mare, ceea ce înseamnă aproape o mie de milioane de operaţii pe secundă (1 Gop/s). Operaţiile logice se pot executa cu rata de 20 Gop/s.

Tab. 3.11 prezintă exemple de valori de performanţă furnizate de simulator. Singura presupunere se referă la perioada ceasului de 100 ns, justificată de simularea electrică a circuitului testat.

(iv) *Configurarea structurilor de date*

Masivul RPA poate fi considerat ca un masiv foarte flexibil de unităţi de calcul la care, odată cu creşterea dimensiunii masivului, numărul biţilor unităţilor de calcul scade. Fig. 3.57 ilustrează acest concept. Trebuie

4 bit | Microcode | Not used | Quit | Edit / Insert / Others
Main Selection | Make-your selection
Simulate | Configure | Display | Keys | I/O | Set Data | Data Window

adder

	CONTINUE	LA0			SP S1	0	0	ACT		
	JSR load	LA1	LD		SP S1	0	0	ACT	Reset D	LD0
	JSR load		LD		SP S1	0	0	ACT	Reset D	LD1
	CONTINUE				SP S1	0	0	ACT	RAM->N 0	LD0
	JSR adder	LA2			SP S1	0	0	ACT	RAM->T 0	LD2
	JSR save		LD	BYT	SP S1	0	0	ACT	RAM->T 0	
	CONTINUE				SP S1	0	0	ACT		
adder	FOR PLN				SP S1	0	0	ACT		
	FOR CYC				SP S1	0	0	ACT		
	NEXT					T∧N	T N	WORD		
	NEXT						N	WORD	Shift T+N	
	RETURN						0	ACT	T>RAMeR	
load	FOR PLN						0	ACT		
	NEXT						0	ACT	Shift T	RD
	RETURN						0	ACT	T>RAMeR	
save	FOR PLN						0	ACT		
	NEXT						0	WORD	Shift T	UR
	RETURN						0	ACT		

Carry save (sum->P)
Carry save (sum->S)
Ripple carry
Ripple carry Save
Climb
Exit

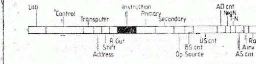


Fig. 3.62 Sistemul de dezvoltare a microcodului RPA, cu menu-ul aferent

observat că masivele dreptunghiulare, ca și cele pătratice, pot fi distribuite direct. De asemenea, este posibilă configurarea unor masive multidimensionale pe RPA, iar dacă numărul elementelor de la margini sînt puteri ale lui 2, orice reorganizare a datelor activate poate fi codificată eficient. Fig. 3.63 prezintă operațiile de deplasare necesare pentru comunicațiile din interiorul unităților de calcul pe 8 biți. Se folosesc biții de reconfigu-

Tabelul 3.11 **Estimări ale performanţei la RPA în cazul a 1024 operaţii. (Toate cifrele reprezintă milioane de operaţii pe secundă). Operaţiile în virgulă mobilă respectă standardul IEEE. Operaţiile în virgulă mobilă în format IBM sînt mult mai rapide.**

Operaţie	Performanţa
adunare de întregi pe 8 biţi	930
adunare de întregi pe 16 biţi	465
adunare de întregi pe 32 biţi	233
înmulţire între întregi pe 8 biţi	128
înmulţire între întregi pe 32 biţi	16
adunare în virgulă mobilă pe 32 biţi	18
înmulţire în virgulă mobilă pe 32 biţi	6
împărţire în virgulă mobilă pe 32 biţi	6
maxim pe 32 biţi	790
operaţii cu şiruri	500—1000

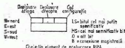
rare pentru asigurarea marginilor pentru operaţiile de deplasare, ca şi a celor de deplasare pe distanţe mari. Registrele de direcţie asigură posibilitatea de a executa deplasări nelaminare. Structurile de date mai complexe, ca şi rotaţiile, pot exploata teoria biţilor muzicali (musical bits — Flanders 1982).

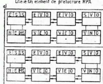
Se poate arăta că, pentru deplasări de tipul vecinătăţii proxime între unităţile procesoare, deşi lărgimea de bandă a comunicaţiilor locale scade cu aproximativ rădăcina pătrată a numărului de procesoare, lărgimea de bandă a comunicaţiilor globale rămîne constantă pentru orice configuraţie. Într-adevăr, lărgimea de bandă a comunicaţiilor pentru structuri de date mai puţin regulate sau în cazul unor deplasări de date mai complexe poate fi mult îmbunătăţită faţă de implementarea soft prin program necesară în cazul unui masiv ne-adaptiv. La o implementare software, poate fi necesară o cantitate mare de repetiţii pentru obţinerea deplasărilor în direcţii multiple, prin mascări şi deplasări alternative. Prin configurarea reţelei de comutare la RPA se realizează concurent diferitele direcţii de deplasare. Mai mult, stivele pot fi folosite condiţional, pentru memorarea datelor cu structuri neregulate, stivele altor unităţi procesoare rămînînd nefolosite.

Fig. 3.63 prezintă codurile necesare în cîmpurile de control local pentru implementarea unor configuraţii regulate care, împreună, dau o idee despre puterea şi flexibilitatea acestei structuri de masiv adaptiv. Cele trei cîmpuri de configuraţie apar la fiecare element procesor. în fig. 3.63 (a) la (c). Acestea sînt codificate cu chei şi corespund stării configuraţiei memorată de fiecare PE.

Fig. 3.63(a) prezintă configuraţia pentru două procesoare pe 8 biţi. Observăm că acestea sînt bucle închise şi execută deci gama completă de operaţii orientate pe „bait".

Pentru efectuarea înmulţirii, este posibil să se configureze procesoarele din fig. 3.63(a) pentru emisia unui bit tuturor biţilor unui procesor dat, simultan la fiecare procesor. Această configuraţie (fig. 3.63(b)) prezintă avantajul unei structuri de magistrală paralelă, unde se pot executa emisii multiple.





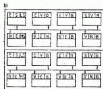


Fig. 3.63 Folosirea cîmpurilor de control local pentru organizarea structurilor de date în vederea prelucrării pe RPA; cheia codifică semnificația cîmpurilor de control. (a) Două microprocesoare pe 8 biți, configurate din cicluri învecinate de PE (observăm că biții MS și LS sînt adiacenți, ceea ce permite efectuarea eficientă a operațiilor de deplasare, ca și transferul transportului). (b) Aceleași două microprocesoare, cu conexiunile necesare creerii unei magistrale locale, pentru operații de emisie de la bitul cel mai puțin semnificativ.

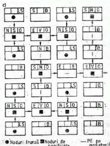


Fig. 3.63 cont. (c) Organizarea unei structuri de arbore binar.

Fig. 3.63 (c) prezintă modul cum poate fi folosit masivul pentru configurarea unui arbore binar de procesoare pe 1 bit, la RPA. În acest caz se folosesc 50% din PE, 25% nu sînt folosite deloc și celelalte 25% sînt folosite pentru configurarea magistralei, transmițînd intrările la ieșiri, deci asigurînd comunicații la distanță mare cînd este cazul.

Structurile mai puțin regulate pot fi distribuite pe masiv prin implementarea unei structuri de comunicații bazată pe pachete, ce folosește rețeaua de comunicații fizică ce are topologia vecinătății proxime, folosind un nivel de microcod. O astfel de implementare a fost programată la RPA, iar rezultatele execuției unui ciclu al acestui algoritm sînt ilustrate în fig. 3.64. S-au implementat pachete de 32 biți, cu două cîmpuri de adrese absolute pentru elementul procesor, x și y. Cu această schemă se poate buferiza într-o capsulă RAM un singur pachet, dar sînt posibile transmisii și recepții simultane de pachete, în absența coliziunilor. Cu posibilitatea de reducere globală implementată în masivul RPA, este posibil să se

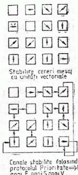


Fig. 3.64 Implementarea protocolului de comutare a pachetelor în rețeaua RPA.

detecteze deterministic o coliziune potenţială în cîţiva micro-cicli şi să se buferizeze pachetele în memoria RAM externă.

Lărgimea totală de bandă a circuitelor de comutare de la RPA este de 2 biţi la intrarea şi 2 biţi la ieşirea fiecărui PE în fiecare micro-ciclu, un total de 4×10^{10} biţi/s, aproximativ echivalent cu CM-1 (vezi §3.5.3), deşi această maşină este de 64 ori mai mare. În modul de comutare al pachetelor, sînt necesari între 100 şi 200 micro-cicli pentru trimiterea a 32 biţi printr-un PE, mai aproape de destinaţie, rezultînd un total de între 3 şi 6×10^{9} biţi/s pentru întregul masiv. Astfel flexibilitatea comunicaţie conduce la scăderea lărgimii de bandă cu un ordin de mărime; totuşi, lărgimea de bandă statică este mare, mult mai eficientă în comparaţie cu o implementare pe un masiv neadaptiv. Acest sistem permite implementarea structurilor de date neregulate şi dinamice.

3.5.5. Transputere şi sisteme înrudite

Transputerul (INMOS 1985) este prezentat în fig. 3.65; este un microprocesor pe o capsulă produs de compania INMOS Ltd. Se distinge de alte microprocesoare prin aceea că a fost proiectat special ca element de bază pentru procesoarele paralele. De aceea, posedă în interiorul circuitului VLSI memorie şi linii de comunicaţie pentru comunicarea cu alte transputere. De aceea, poate fi considerat ca o familie de componente VLSI programabile care pot implementa conceptul de calcul concurent.

INMOS a implementat, în cazul transputerului, atît de multe componente ale unui calculator tradiţional von Neumann, cît s-a putut pe o capsulă de 1 cm², asigurînd în acelaşi timp mecanismele necesare pentru calculul concurent sau bazat pe conceptul de proces. Într-o oarecare măsură, este vorba de a optimiza tehnologia CMOS folosită (vezi §6.1), deoarece comunicaţiile între dispozitivele VLSI au o lărgime de bandă mult mai mică decît cea dintre sub-sistemele din interiorul capsulei. La o capsulă VLSI, pinii sînt scumpi şi relativ lenţi. Mecanismele pentru facilitarea concurenţei includ hardware pentru o coadă de aşteptare pentru procese, cu registre speciale şi instrucţiuni microprogramate care permit crearea task-urilor paralele; un context minimal pentru fiecare proces activ astfel ca comutarea proceselor să fie foarte rapidă; timere hardware; şi un semnal extern de întreruperi.

Partea de execuţie a transputerului este reprezentată de un calculator RISC (cu set redus de instrucţiuni — reduced instructions set computer), care are o viteză mare de execuţie (20 MHz) pentru un număr redus de instrucţiuni şi moduri de adresare. Procesorul ocupă aproximativ 25% din suprafaţă. Alt procent de 25—30% este ocupat de cei 2 KB (T414) de memorie statică, iar legăturile de comunicaţie transputer-transputer ocupă încă 25%. Suprafaţa rămasă este folosită de circuitele de interfaţare pentru sistemul extern de memorie.

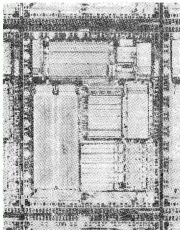


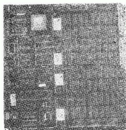
Fig. 3.65 Fotografia circuitului transputerului T800. Capsula conține 4 KB memorie RAM (zona întinsă din stînga jos); controlerele legăturilor INMOS și ale intrării de întrerupere (dreapta jos); o cale de date pe 32 biți și ROM (în mijloc, spre dreapta); și unitatea de calcul în virgulă mobilă și ROM (în partea superioară). Fiecare din aceste blocuri ocupă aproximativ 20% din suprafață; cei 20% care rămîn reprezintă spațiul pentru interfața cu memoria și controlerele externe. (Fotografie pusă la dispoziție de INMOS Ltd.)

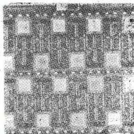
Produsele transputer comercializate includ un procesor pe 32 biți cu o magistrală de adrese/date pe 32 biți multiplexată (T414), un procesor pe 16 biți (T212) și un circuit controler de disc, cu magistrale de date și adrese pe 16 biți și interfață specializată standard. În perioada scrierii lucrării de față, INMOS testa T800, versiunea în virgulă mobilă a transputerului, care posedă în interiorul capsulei o unitate hardware în virgulă mobilă alături de celelalte componente ale lui T414. Folosirea unei tehnologii RAM mult mai dense a permis implementarea la T800 a unei memorii de 4 KB.

La toate produsele, se pot folosi frecvențe diverse, intern pînă la 20 MHz, deși documentația INMOS precizează că o parte a circuitului T800 lucrează la 30 MHz. O caracteristică unică a transputerului este că frecvența externă de ceas este pentru toate componentele de 5 MHz. Astfel, se simplifică proiectarea sistemelor, deoarece frecvența este relativ scă-

zută. Transputerul comunică asincron, deci poate fi combinat cu orice procesor care foloseşte un ceas propriu.

Componentele interne transputerului lucrează concurent; fiecare din cele 4 legături şi co-procesorul în virgulă mobilă pot efectua operaţii utile în timp ce procesorul execută alte instrucţiuni. Fiecare linie de comunicaţie are un canal DMA în sistemul de memorie care va reduce, totuşi un semnificativ, lărgimea de bandă la procesor.





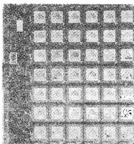


Fig. 3.66 O serie de plăci cu transputere care conţin (a), 1, (b) 16 şi (c) 42 transputere. (Fotografii furnizate de INMOS Ltd.)

Fig. 3.66 ilustrează puterea transputerului T800, prin citeva plăci cu transputere. Fig. 3.66 (a) prezintă o placă cu un sistem cu un transputer şi 2 MB, care atinge 1—2 Mflop/s, fig. 3.66 (b) o placă cu 16 transputere, o componentă care realizează 16—32 Mflop/s, iar fig. 3.66(c) o placă cu 48 transputere care atinge 42—84 Mflop/s! Toate plăcile au formatul dublu eurocard extins. Desigur, într-un caz puterea de calcul este redusă în favoarea memoriei, dar se poate vedea că există un mare interes pentru realizarea unor modele cu o viteză de calcul mare, care utilizează un spaţiu mic de memorie RAM, de exemplu pentru unităţi pipeline. Cu actuala rată de creştere a densităţii de integrare, nu va fi surprinzător să vedem elimina-

rea capsulelor RAM în favoarea unor dispozitive procesor-memorie, al căror emisar este transputerul. Atingerea a 4 Mflop/s şi 64 KB pe o singură capsulă în 1990 nu va fi o realizare surprinzătoare.

(i) *Arhitectura procesorului*

Sistemele cu transputer execută programe scrise în limbajul de programare OCCAM (vezi §4.4.2), care permite descrierea concurenţei între transputerele din sistem sau în cazul unui singur transputer. De aceea, transputerul trebuie să conţină scheme care să faciliteze concurenţa. Un proces executat pe transputer este descris cu 6 registre (vezi fig. 3.67). Cele 6 registre sînt:

(a) o stivă pentru trei operanzi (A, B şi C în fig. 3.67);

(b) un indicator (pointer) al spaţiului de lucru care indică zona de memorie ce păstrează variabilele locale ale procesului;

(c) un indicator pentru adresa următoarei instrucţiuni executate de proces şi

(d) un registru operand, folosit pentru formarea valorilor literale sau a operanzilor.

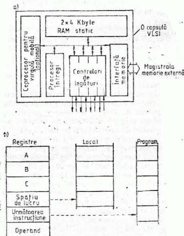


Fig. 3.67 Transputerul INMOS. (a) Arhitectura. (b) Registrele.

Stiva este folosită în manieră convenţională, operaţiile făcînd referinţele implicit la locaţiile din vîrf. Toate instrucţiunile au un format fix şi compact, care respectă principiile arhitecturii RISC, dar care în acelaşi timp permite extinderea setului de instrucţiuni în cadrul aceluiaşi format

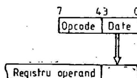


Fig. 3.68 Formatul instrucţiunii la transputer şi registrul operand.

de instrucţiune. De asemenea, se obţine independenţa faţă de lungimea cuvîntului procesor. Formatul prezentat în fig. 3.68 conţine 4 biţi pentru op-cod şi o dată reprezentată pe 4 biţi, care poate fi folosită fie ca operand, fie ca adresă.

Instrucţiunile codificate cu cei 4 biţi includ:

încărcare constantă;
adunare constantă;
încărcare locală;
memorare locală;
încărcare pointer local;
încărcare externă;
memorare externă;
salt;
salt condiţionat;
apel.

Aceste instrucţiuni pot folosi constanta pe 4 biţi conţinută în cîmpul de dată ca valoare literală între 0 şi 15, sau ca adresă relativă faţă de indicatorul spaţiului de lucru pentru referinţe locale, adică ca offset cu valoarea între 0 şi 15. Referinţele externe furnizează un deplasament relativ la vîrful stivei, sau registrul A. Pentru literale cu valori mai mari, două instrucţiuni suplimentare asigură abilitatea de formare a unor date pe baza unor secvenţe de astfel de instrucţiuni pe bait. Toate instrucţiunile îşi încep execuţia prin încărcarea datei pe 4 biţi în registrul operand care va fi apoi şters la încheierea execuţiei funcţiilor respective. Există, în plus, instrucţiuni care încarcă acest registru şi deplasează rezultatul la stînga cu 4 poziţii, cu alte cuvinte realizează un produs cu 16. Aceste instrucţiuni nu şterg registrul operand după execuţie. Două instrucţiuni — *prefix şi prefix negativ* — permit obţinerea în registrul operand a valorilor pozitive sau în complement faţă de 2, pe lungimea cuvîntului procesorului.

Această dată poate fi folosită ca operand de către instrucţiunile directe de mai sus. Statistici efectuate în cadrul analizelor codului generat de compilator indică faptul că instrucţiunile directe conţin operaţiile şi modurile de adresare cele mai des folosite. Mai mult, aceste rezultate arată că cele mai folosite valori literale sînt constante întregi cu valori mici. De aceea, setul simplu de instrucţiuni permite execuţia foarte rapidă a celor mai folosite dintre ele. Oricum, subiectul nu este epuizat, deoarece

chiar şi susţinătorii cei mai înflăcăraţi ai conceptului RISC au nevoie de mai multe instrucţiuni; procesoarele RISC obişnuite au între 64 şi 128 instrucţiuni.

Instrucţiunea de operare (operate) permite interpretarea registrului operand ca un op-cod, realizînd astfel cu un octet, încă 16 instrucţiuni care lucrează cu stiva. Mai mult, instrucţiunea prefix poate fi folosită pentru extinderea domeniului funcţiilor posibile. În mod obişnuit, se pot codifica toate instrucţiunile cu un singur prefix de registru operand. Cele mai folosite operaţii indirecte sînt codificate de tetrada inferioară a primului bait. Ca exemplu, înmulţirea vîrfului stivei cu un operand pe 16 biţi s-ar executa cu următoarea secvenţă de program:

prefix — cei mai semnificativi 4 biţi
prefix — următorii cei mai puţin semnificativi 4 biţi
prefix — ultimii biţi prefix
încărcare constantă — conţine ultimii 4 biţi
prefix — funcţie indirectă
operează — decodifică registrul operand odată

Această secvenţă consumă 4 cicli pentru încărcarea operandului şi 40 pentru execuţia înmulţirii, ceea ce nu este un overhead prea mare în raport cu simplitatea execuţiei. Dacă frecvenţa ceasului este mare, datorită simplităţii decodificării, atunci beneficiile oferite de soluţia transputerului sînt confirmate. De asemenea, secvenţa este compactă necesitînd numai 6 baiţi (doi cicli de magistrală pentru încărcarea instrucţiunii).

(ii) *Mecanismele care implementează conceptul de calcul concurent*

Transputerul acceptă concurenţa în gradul cel mai înalt. Posedă un planificator micro-programat, care permite ca oricît de multe procese să concureze pentru timpul procesorului, singurele limite intervenind la accesul memoriei. Procesele se află în două cozi de aşteptare. Una este pentru procesele active, în curs de execuţie sau care aşteaptă lansarea în execuţie.

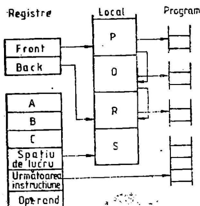


Fig .3.69 Înlănţuirea procesclor în coadă.

Coada proceselor inactive păstrează acele procese care aşteaptă finalizarea unor operaţii de intrare, ieşire sau o întrerupere. Planificatorul nu trebuie să consulte aceste dispozitive, deci nu consumă cicli procesor pentru procesele inactive. Fig. 3.69 ilustrează legăturile specifice unei cozi pen-

tru procese, în acest caz pentru procese active. Două registre memorează începutul și sfîrșitul listei, în timp ce procesul în curs de execuție are propriul indicator al spațiului de lucru și numărător de program încărcat în registrele procesorului. Toate celelalte procese păstrează aceste informații în spațiul de lucru.

Timpii necesari comutării proceselor sînt foarte mici, de ordinul microsecundelor, funcție de instrucțiunea ce se execută. Motivul este faptul că se salvează puține informații de stare. Stiva de evaluare nu trebuie salvată, iar cînd procesul curent nu mai poate continua, numărătorul de program și indicatorul spațiului de lucru sînt salvați în spațiul de lucru, următorul proces fiind preluat din lista activă.

Două microinstrucțiuni asigură mijloacele pentru adăugarea sau ștergerea proceselor în coada de așteptare a proceselor active. Acestea sînt *start proces* și *end proces*. O instrucțiune start proces este necesară pentru fiecare componentă a constructorului PAR din OCCAM (vezi § 4.4.2), iar încheierea corectă a procesului se asigură prin execuția instrucțiunii end proces care, atunci cînd se execută, decrementează un numărător. Numai cînd fiecare componentă a constructorului PAR s-a încheiat, va fi ajuns numărătorul la zero, ceea ce semnifică sfîrșitul corect și continuarea procesului în care a intervenit acest constructor.

(iii) *Legătura IMNOS*

Comunicațiile între procesele de pe un transputer sau între procese de pe diferite transputere se execută cu două instrucțiuni *input mesaj* și *output mesaj*. Comunicația este de tipul punct la punct, nebuferizată. Deci, este necesar un protocol handshake între procese, pentru sincronizare. Aceleași instrucțiuni sînt folosite pentru transmiterea mesajelor între procese ce se execută pe același transputer sau pe transputere diferite. Instrucțiunile folosesc adresa canalului pentru a determina forma comunicației necesare.

Canalele interne sînt reprezentate de un cuvînt în memorie, suficient pentru protocolul handshake stabilit între procesele care comunică. Canalul posedă fie o valoare specială „liber" (empty), sau identitatea unui proces. Un canal este inițializat la valoarea liber, iar cînd un proces este gata să scrie sau să citească, identitatea sa este memorată în locația canalului; apoi, acest proces este scos din coada de așteptare a proceselor active. Cînd și al doilea proces poate participa la comunicație, va găsi identitatea primului în locația de canal, mesajul va fi copiat, după care procesul care așteaptă este adăugat la lista proceselor active. Mesajul este definit de un număr, o locație de canal și un indicator la mesaj. Fig. 3.70 prezintă schematic secvența enunțată anterior.

Canalele externe funcționează într-o manieră similară, cu diferența că interfața execută copierea mesajului prin intermediul liniei de legătură. Fiecare legătură implementează două canale OCCAM în direcții opuse, cu trei fire care propagă semnale de nivele TTL. Comunicațiile pe aceste legături sînt conduse de controlere autonome, care au acces DMA la memoria transputerului. Prin urmare, în timp ce transputerul execută instrucțiuni, se pot executa simultan 4 comunicații bidirecționale. Fiecare controler pentru legătura de comunicație are trei registre, care păstrează

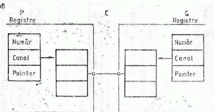


Fig. 3.70 Operarea canalelor de comunicație interne și externe. (a) Primul proces P găsește locația de canal liberă. (b) Procesul este scos din coadă și locația sa este memorată în canal (c). Al doilea proces Q găsește locația lui P în canal, comunicația continuă și procesul P este introdus din nou în coadă.

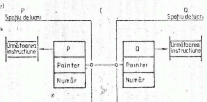


Fig. 3.70 cont. (d) Starea inițială a două canale gata pentru comunicație. (e) Starea în cursul operației de comunicație; se observă că ambele procese sînt scoase din coada de așteptare.

un indicator la zona de lucru a unui proces, un indicator la mesaj şi un număr pe 8 biţi pentru lungimea mesajului, folosit la controlul transferului. Singura diferenţă în operare este că la comunicaţiile externe, ambele procese trebuie scoase din lista proceselor active pe durata cît are loc comunicaţia.

Modul de lucru şi performanţa legăturii de comunicaţie INMOS sînt fundamentale pentru exploatarea efectivă a transputerului. Toate produsele transputer pot executa comunicaţii cu vitezele de 5, 10 şi 20 Mb/s. Deşi transferul este autonom, atît timp cît mesajele transmise au o lungime rezonabilă, o suprapunere totală nu va fi posibilă, deoarece fiecare transfer va consuma o cantitate finită din timpul procesorului pentru scoaterea procesului din coada de aşteptare şi iniţializarea registrelor controlerului. Dacă se măsoară lărgimea de bandă a legăturii de comunicaţie ca funcţie de lungimea mesajului, se poate folosi modelul pipeline clasic cu un timp de iniţializare fără transmisie, urmat de o funcţionare constantă. Prin urmare, granularitatea mesajului modifică lărgimea de bandă. Trebuie să ne reamintim că perioada de iniţializare degradează de asemenea performanţa procesului, în aşa măsură că multe transferuri de mesaje de lungime mică pot satura procesorul.

Fig. 3.71 prezintă dependenţa timpului necesar transmisiei unui mesaj de o lungime dată la un transputer cu legătura de comunicaţie funcţionînd la 10 MHz, şi ceasul procesorului la 12,5 MHz, ca la T414. Rata de transfer

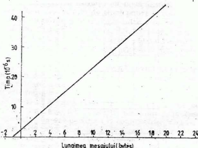


Fig. 3.71 Timpul necesar comunicării pe o legătură INMOS funcţie de lungimea mesajului, pentru un transputer T414 rev. A

maximă sau asimptotică este de aproximativ 0,5 MB/s, ce poate fi realizată în ambele direcţii. Se mai determină lungimea mesajului pentru care se obţine jumătatea performanţei, de aproape un bait. Aceşti parametri vor fi diferiţi pentru procesoare diferite şi legături care folosesc alte viteze de transmisie, timpul de iniţializare fiind proporţional cu ceasul procesorului, iar lărgimea de bandă asimptotică, la un prim ordin, proporţională cu

viteza de transmisie. Astfel, pentru cel mai bun T414 (procesor de 20 MHz și legătura de 20 MHz), se va observa o lărgime de bandă asimptotică de 1 MB/s, și o lungime a mesajului de un bait pentru jumătate din lărgimea de bandă.

Protocolul folosește un pachet de 11 biți, cu un bit de start, un bit pentru diferențiere între date și pachete de achitare, 8 biți de date și un bit de stop. Pachetul de achitare (acknowledge) are 2 biți , un bit de start și unul de stop. Protocolul permite acceptarea unui bait de date de către transputerul receptor, la primirea celui de-al doilea bit (cel care face distincția). În acest mod se pot realiza transmisii continue de pachete de date, în condițiile în care timpul de propagare a semnalului este mic în comparație cu timpul de transmisie a pachetului.

Această pre-acceptare a pachetului de date nu este implementată la transputerele T414 curente, unde nu se va trimite un pachet de achitare pînă ce nu s-a recepționat pachetul complet de date. Acest protocol este implementat la T800, ceea ce permite atingerea unei rate de transfer teoretice maxime de 2 MB/s pe legătură și direcție. Această lărgime de bandă poate fi degradată în cazul traficului bi-direcțional, datorită intercalării pachetelor de achitare cu cele de date.

(iv) *Performanța*

Performanța transputerului este dependentă de un număr de factori, de exemplu frecvența ceasului care poate varia între 12,5 și 20 MHz. De asemenea, dacă datele provin din memoria RAM externă, timpii de acces vor depinde de viteza memoriei și, pentru cele mai multe operații, acesta este un factor major în cadrul vitezei de lucru. În cazul cel mai bun, interfața memoriei externe va avea un ciclu de trei ori mai mare ca al transputerului (150 ns pentru procesorul de 20 MHz). Pe de altă parte, memoria internă necesită un singur ciclu procesor. Este clar, prin urmare, că dacă datele se află în memoria internă se vor obține cîștiguri de viteză. Manualul de referință al transputerului prezintă o evaluare a ciclilor necesari pentru execuția diferitelor construcții OCCAM. Am executat o serie de programe de test pe un sistem cu un transputer T414 rev A, cu frecvența internă a ceasului de 12,5 MHz. Rezultatele obținute sînt prezentate în tab. 3.12.

Tabelul 3.12 **Valori de performanță pentru transputerul T 414 rev A (12,5 MHz). Valorile sînt în milioane de operații pe secundă pentru operanzi din memoria internă sau externă.**

Operația	Performanța	
	Memorie internă	Memorie externă
Adunare cu întregi	1,78	0,23
Înmulțire cu întregi	0,33	0,14
Împărțire cu întregi	0,27	0,13
Adunare în virgulă mobilă	0,03	0,017
Înmulțire în virgulă mobilă	0,03	0,018
Împărțire în virgulă mobilă	0,03	0,016

Cind programatorul folosește limbajul OCCAM, este încurajat să exprime paralelismul, chiar dacă fragmentul de cod este executat pe un singur transputer. Se va încuraja acest stil de programare, în condițiile în care se pot crea procese paralele fără un overhead excesiv. INMOS subliniază că acest overhead este la transputer mai mare ca un apel de funcție dintr-un limbaj secvențial. Pentru a testa acest lucru, am executat codul care urmează pe un transputer (12,5 MHz).

```
VAR b,c
b :=...
c :=...
PAR [j=0 FOR m]
VAR a
SEQ [i=0 FOR n]
a :=b*c
```

S-a executat acest cod cu valori diferite ale lui m și n, produsul lor menținindu-se constant, m*n = 1024. S-a folosit memorie internă și externă, rezultatele fiind trasate în fig. 3.72. Folosind memoria externă, este evident că overhead-urile implicate de crearea proceselor nu devin semnificative pină cind 128 procese paralele execută 1024 înmulțiri cu întregi; iar în cazul memoriei interne, cind 256 procese paralele execută 1024 operații. Aceste cifre corespund la 8, respectiv 4 operații pentru fiecare proces.

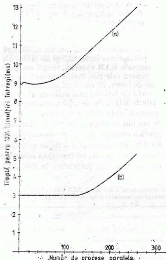


Fig. 3.72 Efectul paralelismului asupra unui singur transputer T 414 rev. A. În acest experiment s-au executat 1024 înmulțiri cu întregi, folosind un număr de task-uri paralele. (a) Datele și programul în memoria externă. (b) Datele și programul ine memoria internă.

Prin urmare, overhead-urile pentru crearea proceselor paralele sint mici. Aceste cifre prezintă și avantajele programării transputerului astfel încît și programele și datele să se afle în memoria internă. Dacă se folosește transputerul în acest mod, este probabil că programul va fi distribuit mai multor transputere. Acesta este denumit paralelism algoritmic, iar tehnica este ilustrată în §4.5.2. În această situație, datele se vor obține prin intermediul legăturilor de comunicație între transputere. Am executat programul de

transputere interconectate pentru a simula efectul dimensiunii pachetelor asupra timpilor de operare. Codul OCCAM a constat dintr-o secvență de înmulțiri între întregi aflați în memoria internă, ambele seturi de operanzi fiind primite de la două canale OCCAM externe, iar

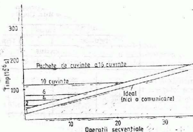


Fig. 3.73 Efectul comunicațiilor asupra operațiilor transputerului. Operanzii și rezultatele pe legăturile INMOS. Fiecare linie prezintă rezultatul pentru o dimensiune a pachetului. Idealul (nici o comunicație) este de asemenea reprezentat.

rezultatele fiind transmise la un canal OCCAM extern. Toate operațiile de I/E au fost buferizate, astfel că operațiile de comunicație să se desfășoare în paralel cu execuția instrucțiunilor de către procesor. Rezultatele sînt prezentate în fig. 3.73, curbele corespunzînd la o dimensiune a pachetului, iar timpul depinde de numărul total de operații executate; curbele sînt comparate cu cea ideală, cînd datele se află în memoria internă.

În prezentarea transputerului în virgulă mobilă, INMOS a afirmat că partea ce funcționează la 20 MHz va lucra cu mai mult de 1,5 Mflop/s. Aceasta înseamnă că va fi un procesor foarte puternic. Aceste afirmații au fost verificate pe eșantioane la Universitatea Southampton. Dacă aplicația permite, transputerul poate fi folosit cu foarte puține circuite suplimentare, ceea ce înseamnă că s-ar putea afla pe o placă format IBM 32 transputere, fără memorie externă. Această placă ar adăuga o cifră impresionantă de 50 Mflop/s la un calculator personal. Tab. 3.13 prezintă timpii de execuție pentru operații în virgulă mobilă format IEEE la T800, publicați de INMOS.

(v) *Construind sisteme cu transputere*

Transputerele pot fi folosite în multe aplicații și în diverse moduri, acum existînd pe piață mai multe sisteme bazate pe transputere, inclusiv rack-uri cu plăci de dezvoltare cu transputere produse de INMOS. Nu contează dacă programul sau datele pot fi partiționate peste o rețea de transputere (vezi §4.5); aspectele de comunicare determină organizarea codului și configurarea legăturilor de comunicație.

Comunicaţia joacă un rol important în toate sistemele bine proiectate. Strangularea comunicaţiei de la un procesor convenţional se află de obicei în interfaţa procesor-memorie, care de regulă este externă procesorului. Nu este surprinzător că operaţiile de comunicaţie pot limita performanţele

Tabelul 3.13 **Timpii de execuţie pentru operaţii în virgulă mobilă la transputerul T800. — 20 este partea care lucrează la 20 MHz, iar — 30 cea care lucrează la 30 MHz. Timpii sînt măsuraţi în ns.**

Operaţia	Partea			
	T800—30		T800—20	
	Precizie			
	Simplă	Dublă	Simplă	Dublă
Adunare	233	233	350	350
Scădere	233	233	350	350
Înmulţire	433	700	650	1050
Împărţire	633	1133	950	1/00

maşinilor paralele, deşi aici limitele intervin la comunicaţia procesor-procesor. Mai mult decît atît, această problemă este fundamentală şi, deşi poate fi deplasată în cadrul ierarhiei sistemului, nu va dispare. De exemplu, putem schimba complexitatea circuitului şi a legăturilor folosite într-un sistem complet comutat contra unei reţele statice cu un diametru mare.

În ultimul caz, cînd se prelucrează o structură de date partiţionată, partajată între procesoarele sistemului, numai în cazul special cînd fiecare sub-est de date este independent, nu vor fi necesare comunicaţii între procesoare. Mai general, datele trebuie partajate între procesoare şi în cazul multor probleme complexitatea comunicaţiilor poate domina complexitatea algoritmului. De exemplu, la sortare sau calculul produselor între matrici, fiecare element al structurii rezultate solicită informaţii de la alte elemente din structurile originale.

Astfel de probleme cu proprietăţi de comunicaţii globale se comportă mai degrabă nefavorabil cu creşterea paralelismului în sistem, dacă conectivitatea între procesoare nu reflectă pe cea dintre sub-structurile de date. Se observă în fig. 3.73 că în cazul comunicaţiilor locale numărul de operaţii (înmulţiri cu întregi) necesar pentru un cuvînt de date recepţionate pe legătura INMOS este 2. Este de asemenea evident că o degradare a lărgimii de bandă a comunicaţiei datorată reţelei cu diametru mare va necesita o granularitate relativ mare la partiţionarea algoritmului. Transputerul poate fi configurat direct datorită celor patru legături ale sale în următoarele reţele : grila bidimensională, cu schimb şi amestec perfect, fluture, alte topologii 4-conexe (vezi §3.3.4) . În toate cazurile, dacă partiţionarea datelor nu corespunde reţelei, atunci lărgimea de bandă se va degrada în legătură cu viteza de calcul a sistemului, proporţional cu diametrul reţelei. Dimensiunile reţelelor de mai sus variază de la $\log_2 n$ la $n_{1/2}$. La aplicaţiile care devin limitate datorită comunicaţiilor, performanţa nu va creşte liniar cu procesoarele adăugate în sistem.

Tehnica care evită scalarea nevaforabilă este de a permite stabilirea între procesoare a unor permutări arbitrare, folosind o reţea în cruce sau

echivalentă. Deşi ultima are multe avantaje, şi permite o modificare arbitrară a paralelismului, costul tinde să crească cu pătratul numărului de procesoare din sistem. Costurile reţelelor fixe variază liniar cu numărul procesoarelor. Totuşi, deoarece transputerul comunică printr-un circuit serial cu o lărgime de bandă mare, costul de interconectare nu este prohibitiv de scump, sau cel puţin nu este aşa pentru masive de pînă la cîteva mii de transputere, care cu T800 ar atinge mai mulţi Gflop/s.

Un sistem cu transputere care permite configurarea unor reţele arbitrare a fost propus iniţial de unul din autori şi este, în momentul de faţă, subiectul unui proiect ESPRIT major în prelucrări avansate de informaţii. Unul din factorii care justifică acest proiect este folosirea transputerelor ca noduri de comutaţie pentru implementarea reţelelor derivate din program, astfel că transputerele să fie configurate în reţele algoritmice. Se creează în mod efectiv un nod mai puternic care are o performanţă de comunicaţie mai mare decît cea a unui transputer. Paralelismul algoritmului poate fi exploatat, cu procesele şi configuraţia reţelei formînd un graf de circulaţie a datelor static sau quasidinamic (vezi §4.5.2. pentru detalii).

Pentru a înţelege argumentele, să considerăm următoarele. Un transputer are puterea de calcul P şi lărgimea de bandă a comunicaţiei C, raportul C/P determinînd cît de curînd lărgimea de bandă a comunicaţiei va deveni insuficientă, odată cu introducerea a noi transputere în sistem. Să luăm un număr mic de transputere, să spunem n, şi să le aranjăm ca în fig. 3.74. S-a arătat (Nicole şi Lloyd 1985) că această configuraţie va implementa orice graf cu n transputere etichetate. Se obţine acest lucru folosind 2 perechi de $2n \times 2n$ comutatoare în cruce complementare. Un astfel de „supernod" poate implementa orice graf 4-conex pentru un anumit algoritm. Să considerăm acum raportul comunicaţiei. Nodul are n transputere, deci o putere nP; de asemenea, are lărgimea de bandă a comunicaţiei nC, deoarece există o singură legătură externă pentru fiecare legătură a unui transputer. Raportul este, prin urmare, acelaşi ca la un singur transputer, C/P. Totuşi, dacă se conectează aceste noduri într-o reţea statică, diametrul reţelei va fi mai mic, deoarece vor fi necesare mai puţine noduri pentru atingerea aceleiaşi performanţe; va fi mai mic cu un factor de n. Există trei efecte asupra scalării comunicaţiilor.

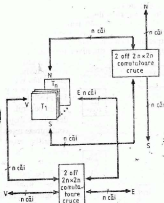


Fig. 3.74 Arhitectura supernodului.

(1) Deoarece nodul are mai multe legături, poate fi interconectat cu mai multe noduri. De exemplu, pînă la $4n+1$ supernoduri pot fi conectate direct cu numai o legătură, $2n+1$ cu două legături, şi aşa mai departe. Din acest motiv, diametrul reţelei produse a fost redus considerabil.

(2) Dacă se implementează o reţea statică cu aceeaşi topologie, diametrul ei va fi redus în orice caz, deoarece sînt necesare mai puţine noduri. De exemplu, la o reţea grilă, diametrul va fi mai mic cu un factor de $n_{1/2}$.

(3) În sfîrşit, deoarece mai puţine noduri pot primi o cantitate mai mare din structura de date, va fi o reducere suprafaţă/volum în cadrul partiţiei structurii de date care va satisface mai bine necesarul de comunicaţii.

Desigur, odată definit un supernod de transputere, se pot construi recursiv sisteme tot mai mari, aşa cum se arată mai jos, cu notaţia din § 1.2.4;

$C_{i+1} = nC_i * 4n^{i-1}S$(2n × 2n reţea în cruce)

$C_0 =$ transputer

O astfel de descriere defineşte un comutator cu mai multe etaje, construit cu noduri în reţea 2n×2n. De exemplu, maşina cu două niveluri construită la Southampton în cadrul ESPRIT este de fapt un set de transputere conectate de o reţea cu trei etaje CLOS. Primul nivel apare în fig. 3.74, al doilea în fig. 3.75, unde fiecare nod este un ciorchine de n transputere sau un supernod.

Supernodul poate fi considerat ca un supercalculator, pentru că respectă modelul de programare OCCAM, acesta asigurînd ordine de transputere pentru fiecare componentă şi instrucţiuni pentru iniţializarea circuitelor de comutare care realizează reţeaua necesară. Astfel, OCCAM este folosit pentru a defini procedura şi structura. Un supernod va conţine în mod obişnuit 10—50 transputere, reprezentînd în sine o staţie de lucru puternică, de sine stătătoare. Pot fi construite maşini mai mari folosind supernodul ca o unitate care se poate multiplica, în masive regulate sau în supernoduri cu un nivel mai înalt.

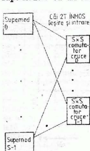


Fig. 3.75

La proiectul ESPRIT, fiecare supernod va avea pînă la 36 transputere, cele mai multe cu numai 256 KB de memorie RAM static (8 capsule), cîte 8 pe o placă cu format triplu eurocard. La nivelul supernodului, unul din transputere, cel de control, va avea capsulele de comutare definite ca dispozitive de I/E şi va fi capabil să configureze reţeaua locală. De asemenea, acest dispozitiv este master pe o magistrală de control de 8 biţi, conectată la toate transputerele din supernod. Această magistrală oferă posibilitatea de a activa şi citi semnale ca reset, analiză şi eroare pe orice transputer şi asigură un mediu de comunicaţie cu lărgime de bandă mică între transputere pentru control şi depanare. De asemenea, această ma-

gistrală permite sincronizarea IF ANY şi IF ALL între transputere, astfel că evenimentele globale să fie semnalate controlerului. Aceasta este necesar pentru utilizarea dinamică a comutatorului pentru a realiza o referință de timp cînd activitatea pe legătură de comunicaţie a încetat şi poate fi reconfigurată. Fiecare supernod de prim nivel va avea RAM şi unităţi de disc.

Controlerul si magistrala sînt conectate la al doilea nivel de un alt controler, care setează comutatoarele externe şi lucrează ca master pe magistrala unde fiecare controler de prim ordin este slave. Un supernod prototip a fost terminat în al treilea trimestru al anului 1987, cu primele exemplare ale transputerului T800. La acest proiect colaborează Apsis SA, Universitatea din Grenoble, INMOS Ltd., Centrul de cercetări al guvernului Marii Britanii RSRE, Universitatea Southampton, Telmat SA şi Thorn EMI plc.

4

LIMBAJE PARALELE

4.1 Introducere

În evoluţia limbajelor de nivel înalt au existat două direcţii. Există limbaje care îşi datorează existenţa unei aplicaţii sau clasă de aplicaţii, ca FORTRAN, COBOL şi C şi cele care au fost proiectate cu scopul dezvoltării ştiinţei calculatoarelor, ca ALGOL, LISP, PASCAL şi PROLOG.

Au existat încercări de a produce un limbaj complet care să încorporeze „cele mai bune" caracteristici ale celor existente, sub forma unui standard. O astfel de încercare a întreprins Departamentul Apărării al S.U.A (DOD) (1978), iar limbajul rezultat, ADA (Tedd et al 1984, Burns 1985) a fost adoptat atît de DOD cît şi de Ministerul Apărării al Marii Britanii. Chiar dacă acum sînt disponibile compilatoare validate, cele mai multe implementări par foarte ineficiente la realizarea concurenţei. Valori tipice pentru crearea şi sincronizarea taskurilor se află în domeniile 4—20 ms, respectiv 1—10 ms (Burns 1985, Clapp et al 1986, Rhode 1986). Aceste valori pot fi comparate cu cifrele similare, dar măsurate în microsecunde, penru OCCAM, limbajul implementat pe transputer.

Se poate afirma că procedura adoptată pentru ADA a fost condamnată încă de la început ; a trecut aproape o decadă de la stabilirea obiectivelor pînă la utilizarea pe scară largă a compilatoarelor validate. Compilatoarele obţinute ocupă spaţiu mare, deoarece limbajul este „complet" şi implementările care acceptă concurenţa reală în cazul sistemelor distribuite sînt încă imature. Mai mult, în cursul acestei perioade de timp s-au produs modificări fundamentale în concepţia şi folosirea calculatoarelor. În particular, această decadă a înregistrat utilizarea pe scară largă a concurenţei. Deşi ADA acceptă concurenţa, există acum mult mai multă experienţă practică privind concurenţa, s-au dezvoltat noi limbaje, ca OCCAM (May şi Taylor 1984), care tratează concurenţa într-o manieră mai simplă, mai consistentă şi la un nivel mai înalt de formalizare. OCCAM nu este un limbaj complet, dar abordarea sa explicită şi minimală îl face un instrument ideal pentru explorarea tehnicilor care exploatează paralelismul.

Totuşi, aşa cum vom demonstra în acest capitol, aplicaţiile numeroase şi foarte diferite ca şi modelele paralele aferente vor necesita structuri de limbaje complet diferite.

Una din diferențierile importante înregistrate în evoluția limbajelor în utlima decadă este cea dintre stilurile de programare imperativ și declarativ. Stilul de programare declarativ a avut efectul cel mai profund asupra cercetărilor privind arhitectura calculatoarelor în această perioadă. Acest stil de programare nu este adecvat arhitecturii von Neumann clasice, cu folosirea intensă a structurilor de date dinamice, care numai pun în evidența deficențele unui singur port de acces la o memorie liniară (ștrangularea von Neumann). Deși această distincție nu este subiectul acestui capitol, există o literatură bogată la dispoziția cititorului interesat. O bună introducere o reprezintă *Funcțional Programming* (Henderson 1980) și *Distributed Computing* (Chamber et al 1984).

O altă direcție importantă care are obiectivul promovării conceptului soft-ului reutilizabil, este reprezentată de programarea orientată obiect (object-oriented). Deși este posibil să cumperi numai anumite circuite integrate, cu specificații precise (date în cataloage), și să le integrezi într-un sistem mai complex, aceeași noțiune a componentelor sfotware standard („off the shelf") s-a limitat pînă acum la pachete de programe științifice (echivalente, să spunem, componentelor MSI TTL). Limbajele orientate obiect oferă programatorului posibilitatea de a construi „capsule software" de complexitate arbitrară; se promovează astfel conceptul „magazinului de programe", unde un inginer poate testa și cumpăra pachete pentru aplicații, fiind sigur (la consultarea cataloagelor) că pot fi integrate într-o aplicație unică. Pentru o introducere entuziastă a acestor concepte, îndrumăm cititorul la lucrarea *Object Oriented Programming* (Cox 1986).

4.1.1 Limbaje imperative

Un limbaj imperativ specifică calculatorului secvența de operații ce trebuie executată sau, dacă sistemul acceptă, secvențe distincte de instrucțiuni ce lucrează concurent. Limbajele imperative au evoluat din primele coduri de programare ale mașinilor, prin abstractizări succesive față de hardware și structurile sale limitate de control. S-au obținut două efecte benefice, creșterea productivității muncii de programare și portabilitatea programelor, obținută prin definirea unui mediu de programare independdent de mașină.

Un limbaj imperativ, chiar la cel mai înalt nivel de abstractizare va reflecta etapele algoritmice pentru obținerea soluției. În plus, față de păstrarea noțiunii de secvență, aceste limbaje mențin conceptul spațiului de adresare liniar, caracteristic celor mai multe mașini. Acesta se reflectă în folosirea masivelor, cu acces direct la toate elementele.

Una din abstracțiunile adăugate mai recent modelului de programare imperativ a fost introducerea concurenței (Harland 1985), caz în care mai multe secvențe de instrucțiuni disjuncte pot fi executate în paralel. Această abstracțiune sau paralelism al fluxurilor de instrucțiuni a apărut înaintea introducerii pe scară largă a concurenței hardware, datorită solicitării de a exploata cît mai intens singura resursă mai costisitoare a calculatorului, unitatea centrală. Task-urile separate logic pot partaja ciclii CPU, evitîndu-se astfel situația cînd un task trebuie să aștepte încheierea unor operații de I/E lente. Se poate obține o partajare nedeterminată a ciclilor CPU prin

generalizarea concurenţei. Orice task care aşteaptă încheierea unor operaţii cu periferice lente poate fi suspendat de sistem, pentru a permite lansarea în execuţie a altui task. Astfel de medii multiprogramate folosesc întreruperile, un mecanism care permite dispozitivelor periferice să atragă atenţia unităţii centrale cînd este cazul.

Lucrarea lui Harland (1985) reprezină o introducere bună pentru conceptul de concurenţă în limbaje de programare imperative. El dezvoltă această temă plecînd de la primele solicitări pentru concurenţă, pînă la cele mai noi, care includ considerarea procesoarelor şi programelor ca valori ale limbajului.

4.1.2 Limbaje declarative

Stilul de programare declarativ este fundamentat matematic şi urmăreşte înlocuirea descrierii algoritmului cu o descriere riguroasă a problemei. De aici denumirea de declarativ. Aceste limbaje se bazează fie pe calculul funcţiilor, calculul lambda, sau pe un subset al logicii cu predicate. Exemple le oferă LISP, respectiv PROLOG. O bună prezentare a aspectelor fundamentale şi a aplicaţiilor în cadrul stilului de programare o oferă cartea *Logic, Algebra and Databases* (Gray 1984). Pentru limbajele declarative, structura de date naturală este lista, care este o aplicaţie indirectă pe un spaţiu liniar de adrese. Oricum, această structură este mai puţin adecvată implementării pe un spaţiu liniar de adrese, decît masivul şi poate conduce la utilizări foarte ineficiente ale celei mai critice resurse a calculatorului, interfaţa procesor-memorie.

Deşi mai puţin eficient decît stilul imperativ de programare,există o cerere continuă pentru limbaje bazate pe un formalism matematic. Motivul este complexitatea crescută a sistemelor de programe moderne şi dependenţa în creştere a societăţii moderne de astfel de sisteme. Deplasarea spre sisteme expert sau inteligente se face pe baza unor stiluri de programare mai puţin regulate şi, deci, mai puţin imperative. Productivitatea programatorului poate fi crescută mult pentru multe aplicaţii prin folosirea limbajelor mai formale (Henderson 1986). Mai mult, deoarece limbajele declarative se bazează pe matematici, este posibilă verificarea formală a sistemelor software create cu ele (Gries 1976).

Un avantaj suplimentar al abordării declarative, sau cel puţin amintit ca avantaj de experţii din acest domeniu, este că astfel de limbaje pot furniza paralelism implicit, la fel cu secvenţierea implicită (Clark şi Gregory 1984, Shapiro 1984). Totuşi există capcane, ca de exemplu generarea paralelă a activităţii neproductive! Cercetarea în acest domeniu este foarte activă, finanţată prin programul de cercetare Alvey, progrese importante fiind de aşteptat pentru anii '90. De exemplu, ICL colaborează la un proiect Alvey important care poate conduce la definirea unor limbaje declarative eficiente, ce se vor baza pe calculatoare paralele. Dacă aceste calculatoare vor fi instrumentele de uz general de la sfîrşitul anilor '90, care vor înlocui calculatoarele tradiţionale, este o problemă deschisă.

Ca o introducere la acest stil de programare, §4.3.4. descrie CMLISP sau * LISP, un limbaj ce se bazează pe LISP, de unde notaţia lambda.

Acest limbaj conţine extensii pentru connection machine care implică o exprimare explicită a paralelismului în structura listei. Connection machine a fost deja descrisă în capitolul 3.

4.1.3 Limbaje orientate obiect

Limbajele orientate obiect se bazează pe două tehnici principale, încapsularea şi moştenirea proprietăţilor. Noţiunea de obiect este o bază mai pragmatic decît riguarea limbajelor logice sau funcţionale. Se asigură o soluţie potenţială la problema de programare discutată anterior, care, în acelaşi timp, poate fi acceptabil de eficientă pe calculatoarele convenţionale. Mai mult, reprezintă un model de calcul care poate fi implementat pe un sistem distribuit.

Din cele două tehnici, încapsularea este cea mai directă şi este folosită adesea ca o tehnică bună de programare (Parnas 1972, Booch 1986) chiar cînd nu este impusă de limbaj. Încapsularea ascunde data şi permite folosirea unor metode sau proceduri de acces la ea. Modulele în ADA (Burns 1985) şi Modula 2 (Wirth 1981) încorporează această tehnică. Accesul la date se asigură numai prin proceduri partajate. Încapsularea datei şi mecanismele de acces definesc, în acest mod, obiectul, iar un limbaj orientat obiect este cel care îndeplineşte acest regim prin construiea unei fortăreţe impenetrabile în jurul acestor obiecte. Aceasta se poate realiza la un nivel înalt, ca la **OBJECTIVE C** şi **ADA** (Cox 1986), sau la un nivel inferior în sistem ca la **SMALLTALK 80** (Goldberg şi Robson 1983).

Avînd încapsulate eforturile unui programator în crearea unor astfel de obiecte, trebuie să se asigure un mecanism pentru evoluţia sau moştenirea acelui obiect, fără a executa un asalt pe scară largă asupra apărării sale. Acestea este a doua tehnică a limbajelor orientate obiect — moştenirea permite programatorului să creeze clase de obiecte, care pot avea mecanisme comune de acces sau formate comune de date. De exemplu, fiind dată o clasă de obiecte „masiv" putem dori să o folosim la implementarea unei sub-clase de obiecte „şir", care se bazează pe obiectele masiv dar au mecanisme de acces specifice şirurilor. Reciproc, fiind dat un set de mecanisme de acces la un obiect, putem dori să extindem domeniul tipului acelui obiect, dar să exploatăm mecanismele de aces care deja există. Acestea sînt exemple de moştenire. Mecanismul de implementare a moştenirii constă în înlocuirea apelului funcţiei sau procedurii la un obiect cu un mecanism ce implică un transfer de mesaje între obiecte. Mesajul reprezintă o cheie prin care se poate selecta mecanismul de acces. Este echivalent cu o construcţie tîrzie sau întîrziată a unui mecanism de acces la un apel de funcţie sau procedură. Astfel, numai în etapa execuţiei, cînd o clasă face o selecţie, se alege mecanismul de acces sau tipul de dată indicat.

Noţiunea de încapsulare poate fi distribuită cu uşurinţă deoarece, cum se arată mai sus, implementările se bazează adesea pe transmiterea de mesaje. Noţiunea de moştenire nu este distribuită la fel de uşor, deoarece foloseşte un arbore al clasei care defineşte şi extinde obiectele. O implementare distribuită a unei clase de obiecte va deveni saturată rapid la rădăcină, dacă se implementează în mod naiv pe un procesor cu structura de arbore. Prin exploatarea paralelismului aplicaţiilor şi multiplicarea struc-

turii clasei (programul) cînd este necesar, se poate obţine o configuraţie eficientă. Mai mult, datorită naturii comunicaţiei, se poate realiza cu uşurinţă o echilibrare dinamică a încărcării. Limbajele orientate obiect pot fi deci considerate ca primi candidaţi pentru exploatarea eficientă a sistemelor paralele, unde „eficient" implică o utilizare eficientă atît a maşinii cît şi a programatorului. Domeniul este nou şi s-au publicat puţine lucrări. Cititorul interesat poate consulta lucrările conferinţei OOPSLA 1986 publicate ca un număr special în notele SIGPLAN (Meyrowits 1986).

Deşi am trecut în revistă cîteva orientări ale limbajelor moderne, acest capitol (cu excepţia CMLISP) respectă direcţiile de dezvoltare a limbajelor imperative, ca şi modul lor de evoluţie pentru a trata diversele aspecte ale paralelismului hardware. Secţiunea 4.3 tratează paralelismul introdus prin structură, aşa cum se întîlneşte la arhitecturile SIMD. Aceste limbaje sînt ideale pentru masive de procesoare, ca ICL DAP şi procesoare vectoriale, ca CRAY-1. Secţiunea 4.4 analizează utilizarea paralelismului procesului sau task-ului întîlnit la si stemele MIMD. Totuşi, exploatarea cea mai obişnuită a paralelismului o asigură compilatoarele vectorizante FORTRAN folosite de cele mai multe calculatoare vectoriale. Această formă de paralelism este extrasă din structurile de bucle din codul secvenţial. Are avantajul(probabil singurul) că programele secvenţiale existente pentru aplicaţii productive pot beneficia de creşterea de viteză de execuţie a instrucţiunilor vectoriale, implemetare la unităţile în virgulă mobilă pipeline. Această soluţie, ca şi dezavantajele ei potenţiale sînt discutate pe larg în § 4.2.

4.2 Paralelism implicit şi vectorizare

4.2.1 Introducere

Limbajul de nivel înalt a fost dezvoltat ca instrument de programare care să exprime algoritmii într-o formă concisă şi independentă de maşină. Unul din limbajele cele mai obişnuite, FORTRAN, îşi are rădăcinile în anii '50 şi, de aceea, reflectă structura maşinilor din acea epocă : calculatoare care execută secvenţe de operaţii asupra unor date scalare. Prin urmare, programarea în aceste limbaje impune descompunerea algoritmului într-o secvenţă de paşi, fiecare constînd în execuţia unei operaţii asupra unui obiect scalar. De obicei ordinea calculelor este arbitrară, de exemplu la adunarea a două matrici este lipsită de importanţă ordinea de combinare a elementelor, deşi un limbaj de programare secvenţial trebuie să implice o anumită ordonare. Aceasta nu numai că măreşte numărul de instrucţiuni, dar poate duce la execuţia ineficientă a algoritmului.

Să considerăm, de exemplu, codul FORTRAN pentru adunarea a

```
DO 10 J = 1,N
DO 10 I = 1,N
 A(I, J) = A(I, J) + B(I, J)
 10 CONTINUE
```

Elementele lui **A** și **B** sînt accesate în ordinea coloanelor, care este prin definiție ordinea în care sînt memorate. La multe calculatoare, dacă această ordine este inversată, programul nu se va mai executa la fel de eficient. În acest exemplu, deoarece algoritmul nu necesită nici o ordonare, este lipsit de înțelepciune să se facă acest lucru prin program. Dacă nu se programează o anumită ordine, compilatorul o va alege pe cea mai eficientă pentru calculatorul țintă. Mai mult, dacă acesta conține paralelism, atunci toate sau o parte din operații pot fi executate concurent, fără nici o analiză sau ambiguitate.

Limbajul APL (Iverson 1962) a fost primul limbaj folosit pe scară largă care exprimă consistent paralelism, deși obiectivul său a fost mai degrabă exprimarea concisă a problemelor decît evaluarea lor paralelă. Iverson a luat concepte și notații matematice pentru a construi un limbaj. Totuși, diferența importantă dintre o descriere matematică a unui algoritm și a unui program care să-l execute constă în descrierea și manipularea structurilor de date. APL conține posibilități puternice de manipulare a datelor.

Capitolele 2 și 3 au descris noi realizări în arhitectura calculatoarelor. Acestea conțin paralelism într-o formă sau alta, regăsit la modul de execuție al instrucțiunilor. Prin urmare, s-a ajuns (aproape două decade după Iverson) la situația de a exprima paralelismul în algoritm pentru a fi executat paralel.

4.2.2 Abordarea ideală

Pentru a ilustra necesitatea enunțată mai înainte, să luăm în considerare dezvoltarea programelor, de la enunțul problemei de rezolvat, pînă la execuția sub forma codului mașină pe calculatorul țintă. Pot fi identificate următoarele 4 etape, unde literele din paranteze indică gradul de paralelism al etapei respective :

(a) alegerea unui algoritm adecvat problemei (P) ;
(b) complicarea programului într-un limbaj de nivel înalt (L) ;
(c) compilarea programului într-un cod direct executabil (O) ;
(d) execuția codului pe mașina țintă (M).

Gradul de paralelism este reprezentat de numărul operațiilor independente ce pot fi executate simultan (vezi § 5.1.2).

În situația ideală, gradul de paralelism nu trebuie să crească odată cu parcurgerea procesului prezentat. Denumim acest principiu conservarea paralelismului, adică

$$P \geqslant L \geqslant O \geqslant M$$

Limbajele de programare care respectă aceste relații permit programatorului să exprime paralelismul implementărilor algoritmilor în termenii unor construcții explicite ale limbajului, fără constrîngeri impuse de mașina țintă. Dacă această expresie a algoritmului se dovedește ineficientă, va trebui aplicată o transformare pentru a ordona calculele. Verificarea dezirabilității acestui principiu este imediată ; înlocuirea unei operații paralele cu o secvență de pași solicită numai o ordonare arbitrară a ope-

rațiilor elementare. Înlocuirea unui proces secvențial cu o operație paralelă impune o analiză mai complexă. Trebuie analizată circulația datelor pentru a se asigura că ordonarea este în fapt arbitrară și că nu există dependențe secvențiale în operațiile paralele propuse. Inevitabil, această analiză este împiedicată de construcții și variabile specifice fazei de execuție.

4.2.3 Vectorizarea

Solicitarea continuă ca noua generație de calculatoare să execute programele deja existente, fără modificări, a condus la un compromis în exploatarea beneficiilor performanței obținute prin paralelism. De aceea, toate calculatorele vectoriale cu succes comercial posedă compilatoare vectorizante (Wolfe 1986).

Cum se compară abordarea prin vectorizare cu idealul nostru?

CRAY-1 reprezintă un exemplu bun pentru această soluție. Programul trebuie scris în FORTRAN Secvențial (L = 1). Totuși, mașina are paralelismul natural 64, deci O = M = 64. În această situație orice paralelism al algoritmului se pierde atunci cînd se exprimă în limbajul de nivel înalt, deci trebuie regenerat de către compilator. Fig. 4.1 ilustrează acest proces : (a) situația ideală și (b pierderea și generarea paralelismului).

Această soluție trebuie considerată numai ca un mijloc temporar de menținere a continuității la transferul pe un calculator paralel. Destul de des, compilatorul vectorizant nu va produce structuri paralele, ca alternativă la vectorizare. Astfel, dezvoltarea noului program trebuie să se facă în întregime în limbajul secvențial, care poate avea efectul întăririi preferințelor pentru programarea secvențială și, mai mult poate conduce la practici de programare obscure. Vectorizarea solicită adesea intervenția programatorului, deoarece multe dependențe pot fi rezolvate în faza de execuție. Rescrierea programului pentru evitarea acestei situații produce adesea construcții obscure și practici de programare rele, care ascund algoritmul de bază și fac întreținerea programului mai dificilă.

4.2.4 Paralelismul mașinii

Din păcate, cele mai folosite limbaje care exprimă paralelism fac acest lucru prin limitarea paralelismului la cel întîlnit pe calculatorul țintă. Deci, L = O = M și se datorește P $\geqslant$ L. Fig. 4.1 (d) ilustrează această relație. Evident aceste limbaje nu sînt portabile, prin definiție. Ele au evoluat în absența oricărui standard de prelucrare paralelă, ca limbajul cel mai simplu și eficient implementat pe o mașină dată. Sînt eficiente deoarece construcțiile lipsite de eficiență sînt complet evitate. De asemenea, programatorul cunoaște paralelismul mașinii și deci poate evita o neconcordanță severă cu structurile de date ale problemei sale. Cu cît se produc mai multe calculatoare (și limbaje) paralele, cu atît problemele portabilității programelor devin mai severe (Williams 1979). Vom aborda ulterior această problemă a portabilității.

4.2.5 Tehnici de vectorizare

Compilatoarele vectorizante analizează programul scris într-un limbaj secvențial,de obicei FORTRAN, și, unde este posibil, generează instrucțiuni paralele sau vectoriale pentru mașina țintă. Pentru a realiza acest lucru, compilatorul trebuie să identifice segmente de cod cu tipare care se știe că sînt vectorizabile. De exemplu, un tipar foarte simplu este:

DO <LABEL> Y = <CONST> TO <CONST>

<LABEL> <ARRAY VAR> (Y) = <ARRAY VAR> (Y) <OP>

<ARRAY VAR> (Y)

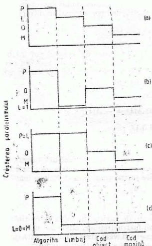


Fig. 4.1 Paralelismul asociat cu procesul de dezvoltare a codului: (a) idealul conservării paralelismului; (b) limbaje vectorizante; (c) limbaje care pot exprima paralelismul problemei; (d) limitare a paralelismului exprimat de limbaje.

Acesta se va înlocui cu una sau mai multe operații vectoriale, funcție de setul instrucțiunilor țintă și diferența între constantele limită ale buclei. Tipare mai complexe conțin bucle definite de variabile, expresii cu masive, instrucțiuni condiționate, apeluri de subrutine și construcții incuibărite.

Procesul poate fi considerat ca o optimizare sau transformare a codului sursă, sau o formă mai compactă a acestuia. De exemplu, compilatorul calculatorului Texas Instruments Advanced Scientific Computer (ASC NX) execută optimizarea asupra reprezentării codului sursă ca un graf orientat. Pentru simplitate vom considera în cadrul acestui capitol numai exemple de transformare a codului sursă. Este evident că locul cel mai probabil unde se întîlnesc secvențe de operații adecvate vectorizării este reprezentat de calculele repetitive, sau buclele DO în FORTRAN. Compilatorul vectorizant va analiza bucla DO, fie cea mai interioară, fie mai multe. Compilatorul ASC NX analizează cel mult trei bucle DO incuibărite și dacă nu există dependențe poate produce o singură instrucțiune mașină. Compilatorul calculatorului BSP analizează buclele DO incuibărite. Dacă una din buclele interioare conține o dependență, le va reordona.

În general, transformarea executată asupra uneia sau mai multor bucle DO este o modificare a secvenței sau ordinii de execuție. La execuția sec-

venţială a buclei DO, ordinea constă în execuţia instrucţiune cu instrucţiune, pentru fiecare valoare a indexului de buclă. Ordinea impusă pentru o execuţie paralelă este una în care fiecare instrucţiune se execută pentru toate valorile indexului, înainte de a se trece la următoarea. Această transformare poate fi executată numai cînd nu există feedback în şi între instrucţiunile buclei. În vectorizare, sarcina principală este detectarea şi analiza acestor dependenţe.

4.2.6 Bariere în calea vectorizării

Este util să trecem în revistă diferitele costrucţii care pot inhiba sau bloca procesul de vectorizare. Unele din ele sînt fundamental nevectorizabile, în timp ce altele sînt construcţii „dificile" pentru analiza întreprinsă de vectorizator.

(*i*) *Instrucţiunile condiţionale şi de salt*

Buclele care conţin instrucţiuni IF şi transfer de control pot inhiba procesul de vectorizare. Totuşi ,multe condiţii codificate cu una sau mai multe instrucţiuni pot fi paralelizate (cu o oarecare pierdere de eficienţă). De exemplu, bucla simplă

```
DO 10 I = 1,N
10 IF (A(I) . LT . 0) A(I) = — A(I)
```

se reduce la construcţia paralelă (§ 4.3.1)

```
A(A. LT . 0) = — A
```

unde A. LT . 0 construieşte vectorul mască care „controlează" operaţia vectorială. Este posibil să se vectorizeze condiţiile reprezentate cu o instrucţiune sau, mai complex, cu mai multe instrucţiuni prin folosirea tehnicii mascării.

(*ii*) *Dependenţe secvenţiale*

Se referă la domeniul cel mai mare de bariere în calea vectorizării. Unele dependenţe se datorează ordinii instrucţiunilor, altele naturii recursive a calculelor. Prima categorie poate fi vectorizată cu uşurinţă prin reordonarea instrucţiunilor şi folosirea zonelor de memorie temporară. Să luăm exemplul simplu :

```
DO 10 I = 2, N
A(I — 1) = NEW(I)
10 OLD(I) = A(I)
```

Această construcţie pare inventată, dar este tipică pentru construcţiile indexate are pot inhiba vectorizarea. Nu este recursivă, deşi aplicarea simplă a transformărilor de ordonare va produce rezultate eronate. Ordo-

narea respectivă va atribui valorile vechi ale lui A(I) vectorului OLD(I); ordinea modificată dă valorile noi ale lui A(I) (de exemplu, NEW(I + 1)) vectorului OLD(I). Problema constă într-o evaluare corectă a timpului de execuție, un bun compilator reordonînd calculele sau asigurînd spațiu de memorie temporară pentru valorile vechi ale lui A(I).

Buclele echivalente de mai jos sînt vectorizabile amîndouă:

```
   DO 10 I = 2,N
   OLD(I) = A(I)
10 A(I - 1) = NEW(I)

   DO 10 I = 2,N
   T (I) = A(I)
   A(I - 1) = NEW(I)
10 OLD(I) = T(I)
```

Construcția recursivă este similară exemplului de mai sus și poate fi exprimată cu o linie, sau implicit cu mai multe atribuiri. Iată două exemple:

```
   DO 10 I = 2, N
   T = A(I - 1) * B(I)
10 A(I) = T + C(I)

   DO 10 I = 2, N
10 A(I) = (A(I - 1) + A(I + 1))/2
```

Acestea sînt exemple de recurențe liniare de ordinul unu, care nu pot fi vectorizate fără a se recurge la algoritmi speciali (vezi § 5.2).

În general, pentru o construcție care are forma

```
DO 10 I = ...
   ⋮
A(I) = A(I + OFSET)
   ⋮
```

unde OFFSET este o variabilă întreagă, vectorizarea este imposibilă. Compilatorul nu cunoaște în faza compilării că OFFSET are același semn cu incrementul buclei, sau nu intervine nici o suprapunere între membrul stîng și cel drept din instrucțiunea de atribuire. Astfel, construcția poate fi teoretic vectorizată, dar practic niciodată, dacă nu se execută compilări de test corespunzătoare fazei de execuție.

(*iii*) *Indexare neliniară și indirectă*

Vectorizarea poate fi înhibată, de asemenea, de anumite expresii cu indecși. Combinații liniare simple de variabile indexate în cadrul buclelor nu ar trebui să producă probleme, atît timp cît nu există restricții hard-

ware. Dacă, însă, expresiile sînt neliniare sau conțin referințe indirecte, vectorizarea nu mai este un proces banal. Iată exemple de expresii cu indecși care pot inhiba paralelizarea :

$$(I * J + K)$$

$$(IV(I))$$

În primul caz dacă fie I, fie J sînt invariante în bucla considerată, expresia este liniară și vectorizabilă, altfel nu.

(iv) Apeluri de subrutine în cadrul buclelor

Ultima barieră în calea vectorizării, luată aici în considerare, este includerea subrutinelor sau a apelurilor de funcții definite de utilizator, în cadrul buclelor. Acestea nu sînt vectorizabile deoarece, de obicei, subprogramele se compilează separat și compilatorului îi lipsește informația necesară analizei.

4.2.7 Vectorizatorul BSP

Compilatorul BSP (Austin 1979) este un exemplu bun de utilizare a tehnicilor de vectorizare. Se bazează pe experiența în acest domeniu de la Universitatea Illinois. El analizează mai mult de un nivel de bucle încuibărite și poate executa anumite reordonări a instrucțiunilor și buclelor DO. În plus, vectorizează buclele care conțin recurențe liniare de prim ordin și anumite bucle cu instrucțiuni IF. Următoarele exemple de vectorizare sînt preluate din lucrarea lui Austin (1979). Vectorizarea se exprimă în termenii construcțiilor paralele BSP.

În acest ultim exemplu, a fost izolată o recurență în J și sau vectorizat buclele cea mai interioară și cea mai exterioară, în M și L. Observăm că, deși sistemul hardware BSP poate manipula recurențe de ordinul 1, vectorizarea recurențelor va fi utilizată numai dacă nu se poate realiza altă vectorizare a unui set dat de bucle DO. Motivul este că evaluarea paralelă a recurențelor liniare de ordinul 1 nu este 100% eficientă (vezi § 5.2).

4.3 Paralelismul structural

Secțiunea 4.2 furnizează o soluție la problema portabilității, care constă în utilizarea în continuare a limbajelor existente (secvențiale), iar generarea structurii problemei sau algoritmului pentru sistemul hardware respectiv este lăsată în sarcina sistemului. Tehnicile pentru generarea automată a paralelismului sînt limitate la optimizarea buclelor pentru execuția lor pe procesoare vectoriale. Exploatarea automată a altor forme de paralelism,de exemplu a sistemelor cu multplicarea resurselor, nu este pe deplin înțeleasă, rezultatele obținute arătînd o utilizare slabă a procesorului și resurselor de comunicație. Una din problemele importante în această privință este lipsa formalismului pentru descrierea structurii problemei în cadrul unui model consistent de calcul care va facilita o organizare a acelei

```
C           LOOP CONTAINING AN IF STATEMENT
              DO 2 I=1,N-1
                IF (M,GT,3) A(I) = A(I+1) + C(I)
2           CONTINUE
C           IF LOOP VECTORIZED
                 WHERE (M,GT,3) A(1:N-1) = A(2:N). +C(1:N-1)
C           MULTI-STATEMENT CONDITIONAL
             DO 3 I=1,N
                IF (A(I),LT,B(I)) GOTO 31
                  C(I) = E(I) + 2
                GOTO 3
31                C(I) = F(I)
3           CONTINUE
C           MULTI-STATEMENT LOOP VECTORISED
                LOGICAL L(N)
                L = A(1:N) , GE , B(1:N)
                WHERE (,(L) DO
                   C(1:N) = E(1:N) +2
                   A(1:N) = C
                OTHERWISE
                   C(1:N) =F(1:N)
                ENDWHERE
C           MULTI-STATEMENT LOOP WITH NON-VECTORISABLE RECURRENCE
              DO 4 I=2,100
                 A(I) = B(I) + 1
                 C(I) = D(I-1) + E(I)
                 D(I) =C(I) + A(I)
4                F(I) = D(I) + 2
C           PARTIAL VECTORISATION
              A(2:100) = B(2:100) + 1
              DO I = A,100
                 C(I) = D(I-1) + E(I)
                 D(I) = C(I) + A(I)
              END DO
              F(2:100) = D(2:100) + 2
C           LOOP REQUIRING STATEMENT REORDERING
              DO 5 I = 1,N
                 A(I) = B(I) + C(I)
                 C(I) = B(I-1)
5                B(I) = 28*A(I+1)
C           VECTORISED WITH TEMPORARY AND REORDERING
              REAL TEMP(N)
              TEMP(1:N) = A(2:N+1)
              A(1:N) = B(1:N) + C(1:N)
              B(1:N) = 2*TEMP(1:N)
                 C(1:N) = B(0:N-1)
C                LOOP REQUIRING REORDERING
                   DO 6 I = 1, N
                   DO 6 J = 1,N
                   DO 6 K = 1,,L
                      A(I,J+1,K) + B(I,J-1,K)
6                     B(I,J,K) = A(I,J,K)
C                VECTORIZATION WITH REORDERING
                   DO 7 J = 1,,N
                      A(1:M, J+1, 1:L) = B(1:M, J-1, 1:L)
7                     B(1:M, J, 1:L) = A(1:M, J.1:L)
```

structuri corespunzător structurii mașinii. Această problemă nu este definită bine în limbaj, care trebuie să folosească bucle și indexări pentru a o exprima. Alte abordări mai teoretice tratează obiectele masiv și clasele de aplicații ce li se pot aplica (de exemplu, vezi §3.3 și Flanders (1982)).

Chiar proiectat pentru un calculator vectorial, compilatorul vectorizant nu este soluția ideală pentru portabilitate. Desigur, programatorul va optimiza codul FORTRAN secvențial, astfel încît compilatorul să vectorizeze cît mai mult posibil. Datorită diferențelor hard-ului vectorial, lungimea optimă a buclelor și modalitățile de acces la memorie se vor schimba de la mașină la mașină. Prin urmare, soluția vectorizării pentru asigurarea portabilității cade, deoarece programatorul tratează FORTRAN-ul, ca și în trecut, ca un limbaj de asamblare. O soluție alternativă constă în declararea explicită de către programator a părților de program care să fie executate în paralel. În acest mod, se poate alege structura paralelă care să exprime soluția problemei, fără referire la structura mașinii folosite. Acum problema portabilității devine una de implementare, deoarece expresiile paralele folosite de programator trebuie să se execute eficient pe mașina țintă.

Pentru exprimarea explicită a paralelismului sînt disponibile două tehnici : prin folosirea unei descrieri a structurii de date care să exprime paralelism , *paralelismul structural*, sau prin folosirea unei descrieri a programului sau a structurii procesului pentru exprimarea paralelismului, *paralelismul procesului*. În unele circumstanțe cele două metode pot fi foarte similare, deoarece structura procesului unui program distribuit poate fi proiectată să exploateze structura datelor. Pentru a face o distincție în aceste circumstanțe, să presupunem că paralelismul structurii este definit ca granularitatea unei singure operații și că operațiile se execută simultan cu fiecare element al structurii de date. Paralelismul procesului, pe de altă parte, se definește cu o granularitate mare, cu un flux de instrucțiuni și o stare asociată fiecărui element, sau (mai probabil) cu fiecare partiție a structurii de date.

Între aceste forme de paralelism, distincția o face modelul de calcul corespunzător sistemului hardware, ca și maniera în care este echilibrată încărcarea sistemului. La paralelismul structural, se pot considera procesoare virtuale asociate cîte unul la un element structural de date, datele activate fiind distribuite pe procesoarele disponibile într-o manieră care echilibrează încărcarea acestora. Astfel, sarcina de lucru este împărțită prin redistribuirea elementelor structurale de date — data remapping. De exemplu, dacă s-a selectat pentru procesare numai o linie a unei matrici distribuită, poate fi necesară o redistribuire a elementelor liniei pentru a încărca toate procesoarele. Această metodă poate fi interpretată și ca o redistribuire a procesoarelor virtuale la procesoarele existente pentru a le menține încărcate. La paralelismul procesului, pe de altă parte, partiția programului sau procesului este virtualizată și se realizează echilibrul încărcării prin distribuirea proceselor pe procesoare. Paralelismul procesului se tratează mai pe larg în §4.4 : această secțiune tratează numai exploatarea paralelismului structural.

În esență , paralelismul structural este o metodă formală pentru exprimarea rezultatului transformărilor de vectorizare prezentate în § 4.2, deoarece semantica operațională a procesorului vectorial corespunde acestui tip

de paralelism. Vectorizarea se poate folosi în cazul calculatoarelor vectoriale, deoarece accesul în maniera pipeline la un sistem de memorie unic poate masca transformarea datelor, implicită în cazul acestei semantici şi, de asemenea, deoarece eficienţa unui pipeline este o funcţie asimptotică de lungimea vectorului (vezi § 1.3).

Paralelismul structural s-a folosit la mai multe extensii de limbaje, de obicei pentru exprimarea paralelismului SIMD caracteristic masivelor de procesoare, ca în cazul DAP FORTRAN (ICL 1979 a). Totuşi, aceste limbaje au tins să exprime paralelismul sistemului hardware, lăsînd programatorului sarcina unei încărcări ridicate a procesorului. De aceea, pot fi considerate limbaje de nivel scăzut, specifice maşinii. Viitorul standard FORTRAN, FORTRAN 8X (ANSI 1985) cuprinde extensii care vor permite manipularea oricărui masiv ca o structură paralelă de date. Această propunere este binevenită, deşi apare cu întîrziere şi incompletă; totuşi, va permite programatorului să exprime structura şi paralelismul inerent unui algoritm, mai degrabă decît cele ale sistemului hardware. Vom trata acest standard ulterior în această secţiune.

Folosind paralelismul structural, programatorul poate folosi, virtual nelimitat, paralelism pentru exprimarea unui algoritm. Limbaje generale complete de acest tip permit tratarea structurilor de date ca obiecte asupra cărora se pot executa operaţii paralele. Poate fi interpretat ca o alocare a unui procesor virtual la fiecare element al structurii de date (de exemplu, o linie a unei matrici), sau implicit printr-o construcţie condiţională a limbajului (ca WHERE în cazul FORTRAN 8X). Compilatorul, sau sistemul, în faza de execuţie, va aloca procesoare virtuale procesoarelor reale din sistem.

Ideea paralelismului nelimitat poate deveni stînjenitoare în cazul abordării pe bază de procese, unde overhead-ul creerii multor fluxuri de instrucţiuni poate fi mai mare decît volumul propriu-zis de lucru. Atît la paralelismul structural cît şi al procesului, în faza de compilare se poate realiza o transformare de la paralel la secvenţial pentru o anumită arhitectură, pentru a optimiza performanţa. Totuşi, în cazul paralelismului procesului, transformarea nu este atît de directă şi este improbabil ca un sistem să realizeze acest lucru în faza de execuţie. În cazul paralelismului structurii de date, transformarea este simplă, numai o ordonare a seturilor de operaţii similare. Mai mult, orice ordine impusă poate optimiza modurile de acces la memorie (Jesshope 1984). Este adevărat că necesarul de memorie poate fi mare, datorită necesităţii de a executa operaţiile ca şi simultan. De exemplu, expresiile complexe cu masive vor necesita spaţii de memorie temporară. Optimizările, ca şi cele folosite pentru vectorizare pot reduce necesitatea de a corespunde cît mai mult paralelismului sistemului hardware ţintă.

4.3.1. Construcţii de tip masiv

Vom lua în considerare un număr de construcţii care pot fi folosite pentru exprimarea paralelismului structurii prin utilizarea masivelor. Se bazează pe limbajul FORTRAN, dar sintaxa nu reflectă cu necesitate o implementare în curs, sau propusă. Se încearcă deducerea unei extensii

consistente a unui limbaj, care permite exprimarea naturală a paralelismului structural al masivelor. Aceste presupuneri diferă în unele aspecte importante de cele propuse de Comitetul ANSI X3J3, care, printre altele, a luat în considerare extensii de tipul masiv pentru următorul standard FORTRAN (8X). Propunerile ANSI sînt tratate pe larg în § 4.3.3.

Datorită nivelului mai înalt de abstractizare întîlnit la construcţia de tip masiv, în comparaţie cu un cod secvenţial, ne-am aştepta să vedem o reducere a codului necesar pentru exprimarea unui algoritm dat. Se poate face analogie cu expresivitatea algebrei matricilor faţă de cea a scalarilor. Prin urmare, orice construcţie inclusă în limbaj trebuie să reflecte acest lucru şi să permită o exprimare concisă a operaţiilor între obiecte de date evaluate ca masive. APL implementează în mod extrem această filozofie, ceea ce determină obţinerea unor programe greu de citit. Totuşi, nu dorim să adăugăm construcţiilor secvenţiale existente în mod simplist o sintaxă stînjenitoare. Cealaltă extremă poate fi ilustrată de PAR DO, propusă ca alternativă la instrucţiunea FORTRAN DO, caz în care se modifică ordinea de evaluare astfel ca fiecare instrucţiune să fie executată pentru toate valorile incluse, în DO, înainte de a trece la execuţia următoarei instrucţiuni. Astfel, corpul buclei se execută în secvenţă, dar cu fiecare instrucţiune executată pentru întreg domeniul de valori indicat de DO, ca şi cum fiecare instrucţiune ar fi inclusă într-o buclă proprie. Sintaxa este confuză şi prolixă, şi este mult mai natural de folosit bucla DO corespunzătoare structurii operanzilor masiv.

(*i*) *Masive ca obiecte elementare de date*

În acest caz masivele sînt considerate ca obiecte de date distincte, de un anumit rang (dimensionalitate). Prin urmare, orice referinţă la masiv va implica întregul masiv. Trebuie să remarcăm că această abordare nu elimină posibilitatea folosirii masivului ca o mulţime de obiecte secvenţiale deoarece se poate folosi indexarea pentru reducerea rangului obiectului, prin selecţie. Astfel, de exemplu, fiind dat un masiv tri-dimensional A, se spune că are rangul 3, iar orice referinţă la numele A singur va implica o referinţă la toate elementele în paralel. Se poate lua în considerare indexarea unei dimensiuni a lui A ca o operaţie de selecţie, care reduce rangul lui A cu 1. Dacă toate dimensiunile sînt indexate, din A se selectează un scalar sau obiect de date de rang 0. În continuare se prezintă mai multe detalii privind mecanismele de selecţie. Trebuie să notăm că acest concept are un precedent în limbajele secvenţiale; de exemplu, în FORTRAN, referinţa la un număr de masiv în cadrul instrucţiunilor READ sau WRITE implică referinţa la toate elementele masivului.

Deşi această abordare este cea mai generală, multe extensii de limbaje au făcut un compromis, în aceea că numărul de dimensiuni ale unui masiv, considerat ca obiect paralel, este limitat. Alte dimensiuni trebuie indexate în cadrul corpului programului, asigurîndu-se astfel mulţimi de obiecte masiv paralele cu un număr limitat de dimensiuni. Acest compromis s-a adoptat aproape exclusiv la masivele de procesoare, unde numărul de dimensiuni care pot fi referite în paralel corespunde dimensiunii şi formei sistemului hardware. Are avantajul diferenţierii acceselor paralele şi secvenţial întîlnit la memoriile masivelor de procesoare, dar şi dezavantajul dependenţei de maşină, şi deci, a lipsei de portabilitate.

Exemple ale acestei soluţii le întîlnim la CFD (Stevens 1975), GLYPNIR (Lawrie et al 1975) şi ACTUS (Perott 1979), toate limbaje propuse sau implementate pentru ILLIAC IV, CFD şi GLYPNIR manipulează masive de 64 elemente cu o dimensiune, care corespund masivului de procesoare de la ILLIAC.ACTUS este mai general, dar implementarea originală de pe ILLIAC IV permitea referirea în paralel numai a unei dimensiuni a unui masiv PASCAL. Un alt exemplu poate fi întîlnit la DAP FORTRAN (ICL 1979), unde pot fi referite în paralel fie prima, fie primele două dimensiuni ale unui masiv. Din nou, cele două dimensiuni trebuie să corespundă dimensiunii DAP. Atît DAP FORTRAN, cît şi CFD sînt tratate în § 4.4.

(ii) Selecţia din obiectele masiv

Se presupune că un masiv este definit ca mai sus, ca un obiect paralel de date şi că mecanismele de indexare sau tehnicile de selecţie reduc rangul. Astfel sub-setul de obiecte masiv care pot fi la rîndul lor obiecte paralele masiv, poate fi manipulat în cadrul limbajului. Există multe tehnici pentru realizarea selecţiei, dintre care unele pot avea semnificaţii conflictuale.

Selectarea obiectelor cu rang redus. Acest mecanism este probabil cel mai important deoarece asigură compatibilitate în sens descrescător cu limbajul secvenţial de bază. În forma cea mai simplă este echivalent indexării din limbajele secvenţiale. Diferenţa constă în aceea că fiind dat un obiect masiv de rang n, să spunem, toate obiectele masiv cu rang redus sînt tot elemente atomice ale limbajului. De exemplu vectorii linie, vectorii coloană, ca şi scalarii sînt obiecte atomice valide, selecţii dintr-un masiv bi-dimensional sau matrice. Vectorii linie sau coloană se selectează prin indexarea unei dimensiuni a masivului, iar scalarul prin indexarea ambelor dimensiuni ale masivului. Astfel, în FORTRAN, fiind dat un masiv sau matrice **A** bi-dimensional NxN, atunci

(a) **A** va referi matricea întreagă,

(b) **A(I,)** va referi linia I a lui **A** (un vector),

(c) **A(,J)** va referi coloana J a lui **A** (tot un vector),

(d) **A(I, J)** va referi, cum era de aşteptat, elementul I, J al lui **A**(un scalar).

La unele limbaje, sintaxa acestui mecanism de selecţie este accentuată cu un simbol special, de obicei un asterix, plasat în poziţia unde lipseşte indicele. Acest simbol poate fi interpretat ca un selector al întregii dimensiuni. Astfel, exemplele anterioare devin :

(a) **A** sau **A** (*, *)

(b) **A** (I, *)

(c) **A**(*, J)

Datorită eleganţei şi simplităţii sale, se va folosi eliminarea indicelui în toate exemplele ce urmează, dacă nu se va folosi o sintaxă specifică.

Selectarea unei game de valori. Adesea va fi necesar să selectăm o gamă de valori corespunzătoare unui indice. Astfel, în loc de a reduce rangul unui obiect masiv, se reduce dimensiunea sa. Gama trebuie să specifice , prin urmare, o submulţime a domeniului întregii dimensiuni. Aceasta se poate defini, ca în controlul buclei, de o pereche sau triplet de întregi sau expresii

cu întregi. Se va specifica un index de start, un index opțional, corespunzător pasului și un index limită, folosit ca selector în una din dimensiunile masivului. Astfel, pentru aceeași matrice N × N, **A**(2 : N−1 ; 2 :N−1) va selecta primele trei elemente ale liniilor impare. În ambele exemple, se folosește sintaxa start : (pas) limită, unde dacă se omite, pasul se presupune a fi unitatea. Aceste mecanisme de selecție cu reducerea rangului sau a domeniului sînt ilustrate în fig. 4.2. pentru un masiv 8 × 8.

Selecția cu folosirea masivelor întregi. Mecanismele de selecție simple descrise mai sus au ca un caz particular mult mai familiară indexare secvențială. Se poate afirma că același este adevărul și pentru indexarea cu masive de întregi și că este un caz special al indexării întîlnită în limbajele secvențiale. Totuși, există cel puțin două moduri în care poate fi folosită indexarea indirectă, și ambele pot fi reprezentate cu aceeași sintaxă.

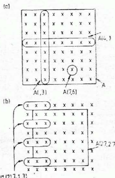


Fig. 4.2 Un exemplu de selecție dintr-un masiv 8×8 : (a) selecții prin reducerea rangului ; (b selecții cu reducerea domeniului.

O interpretare corespunde unei organizări liniare sau unui produs cartezian a unor structuri liniare peste mai mult de o dimensiune. Se prezintă acest lucru în continuare, prin folosirea unor construcții secvențiale, unde **IV** și **JV** sînt vectorii întregi care definesc structurile :

(a) **A**(I, **JV**(J))
(b) **A**(**IV**(I), **JV**(J))

Aici (a) reprezintă o structură liniară ce corespunde celei de-a doua dimensiuni a lui **A**, iar (b) reprezintă produsul cartezian al structurilor liniare în ambele dimensiuni ale lui **A**. Se poate vedea că dacă s-ar aplica în paralel, cu elidarea indicelui

(a) **A**(,**JV**)
(b) **A**(**IV**, **JV**)

atunci masivele rezultate vor avea același rang cu **A**, dar vor avea dimensiunile selectate după dimensiuni corespunzător vectorului întreg. Structurile produse pot conține copii ale unui aceluiași element ; totuși, valorile elementelor vectorului care definește noua structură trebuie să se afle în domeniul de definiție al dimensiunii respective. Astfel, dacă **A** este un masiv N × N și **IV** și **JV** sînt vectori cu domeniul M, atunci toate elementele lui **IV** și **JV** trebuie să fie mai mici sau egale cu N, (a) va produce un masiv N × M, iar (b) un masiv M × M. Ambele sînt structuri arbitrare cu elemente din **A**.

Cealaltă interpretare a acestei construcții se referă la o proiecție pe una sau mai multe dimensiuni ale unui masiv. Aceasta este o operație de reducere a rangului, prezentată în continuare folosind construcții secvențiale similare. **JV** este în acest caz un vector întreg, iar **JM** un masiv întreg,

ambele definind o zonă din masiv care este proiectată pe dimensiunile rămase:

(c) **A**(I, **JV**(I))
(d) **B** (I, J, **JM**(I, J))

Aici este important de notat că vectorul index **JV** şi masivul index **JM** au aceeaşi formă cu zona de masiv pe care o selectează şi sînt indexate în acelaşi mod. Dacă se foloseşte eliminarea indicelui, nu se poate diferenţia între cele două moduri de indexare indirectă, deoarece (c) şi (d) produc următoarele construcţii paralele

(c) **A**(,**JV**)
(d) **B**(,,**JM**)

Se poate vedea că sintaxa construcţiei paralele definită de (c) este identică cu cea definită de (a), dar ele au semnificaţii complet diferite. Dacă **A** este un masiv N × N, iar **B** un masiv N × N × N, atunci **JV** trebuie să fie un vector cu N elemente, iar **JM** un masiv N × N. Ambele reduc rangul masivelor pe care le indexează ca şi cum ar fi scalari; totuşi, felia (slice) obţinută nu este lamilară ci proiectată pe dimensiunile elidate. Din nou, valorile elementelor masivului index trebuie să fie mai mici sau egale cu domeniul corespunzător dimensiunii de unde se selectează. Aceste două tehnici sînt ilustrate în fig. 4.3, pentru un masiv 4 × 4.

Se va arăta mai tîrziu că varianta de structură obţinută prin proiecţie, cînd se foloseşte împreună cu altă construcţie, poate emula tehnica generală de mapare. Alegerea evidentă a semanticii pentru această construcţie este, prin urmare, cea a unei proiecţii. Această convenţie este adoptată la toate exemplele ulterioare, cînd nu se fac alte menţiuni. Această tehnică de selecţie poate fi folosită împreună cu alte tehnici de indexare pentru alte dimensiuni ale masivului, inclusiv alte masive index. Totuşi, toate masivele indexate trebuie să se conformeze formei masivului produs. Astfel, TABLE (,,I,J,K) este o selecţie validă din TABLE, chiar dacă I, J şi K sînt masive întregi. În general, I, J şi K trebuie să fie scalari întregi, fie masive întregi care se conformează structurii produse (de exemplu, acele dimensiuni în care este implicată gama întreagă). Conceptul de conformitate este tratat mai pe larg ulterior.

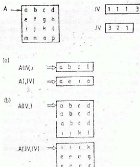


Fig. 4.3 Un exemplu de selecţie dintr-un masiv A folosind vectori întregi (**IV** şi **JV**): (a) selecţia prin proiecţie; (b) o aplicaţie selectată după una şi, apoi, două dimensiuni ale masivului

Selecţie folosind obiecte masiv boolean. Un alt mecanism de selecţie care se poate aplica obiectelor masiv foloseşte masive sau expresii logice. Selecţia se face conform valorii de adevăr a selectorului logic. Astfel, ele-

mentul i poate fi selectat dintr-un masiv uni-dimensional prin indexarea lui cu un masiv boolean uni-dimensional al cărui element i are valoarea „adevărat". Desigur, masivele trebuie să fie conforme. Similar, s-ar putea folosi același masiv boolean pentru selecția fie a unui vector linie sau coloană dintr-un masiv sau matrice bi-dimensională. Aceste exemple apar în fig. 4.4.

Toate exemplele de mai sus folosesc această formă de indexare ca un mecanism de selecție cu reducerea rangului. DAP FORTRAN (vezi § 4.3.2) insistă asupra acestei restricții. Totuși, este posibil, deși nu este de dorit, de extins această formă de indexare astfel încît să determine reducerea rangului. În acest caz, vectorul boolean ar avea un număr de elemente cu valoarea adevărat, iar selecția va produce un obiect cu același rang, dar numai cu acele elemente ce corespund valorii „adevărat". Se pot folosi tehnici similare pentru actualizarea selectivă a unui obiect masiv. Și aceste elemente se vor discuta în detaliu ulterior.

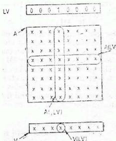


Fig. 4.4 Un exemplu de selecție dintr-un masiv 8×8 folosind un vector logic LV.

Indexarea cu deplasare. Strict vorbind, nu este un mecanism de selecție; totuși, deoarece este adesea implementat ca o tehnică de indexare, am inclus-o în această secțiune. Este un mecanism de aliniere folosit pentru deplasarea sau rotația obiectelor masiv după o direcție dată. Un exemplu bun de folosire îl oferă tehnicile de relaxare a caroiajelor (vezi § 5.6.1), unde fiecare punct al caroiajului este actualizat pe baza unei medii a punctelor vecine. Cazul cel mai simplu este cînd punctul ia valoarea medie a celor 4 vecini mai apropiați după două dimensiuni. Cu sintaxa folosită de DAP FORTRAN, se poate scrie:

$$\mathbf{A} = (\mathbf{A}(+,) + \mathbf{A}(-,) + \mathbf{A}(,-) + \mathbf{A}(,+))/4$$

Se actualizează simultan toate elementele lui **A**, pe baza elementelor vecine, selectate prin deplasarea masivului la stînga (+) și dreapta (−) după fiecare dimensiune. Pentru valorile interioare ale lui **A**, codul secvențial echivalent este DO 10 I = 2, N − 1

DO 10 J = 2, N−1

10 ANEW (I, J) = (A(I + 1, J) + A(I−1, J) + A(I, J−1) + A (I, J + 1))/4

DO 20 I = 2, N − 1

DO 20 J = 2, N − 1

Modul cel mai natural de scriere secvențială ar fi prin înlocuirea lui ANEW (I, J) cu A(I, J) în bucla 10. Totuși, aceasta este recursivă și la evaluare ar folosi atît valori vechi cît și noi ale lui **A**, definite prin ordinea din buclă. Specialiștii în analiza numerică vor recunoaște diferența dintre relaxările

Gauss-Seidel şi Jacobi. Se va vedea în § 4.3.3 că FORTRAN 8X propune implementarea acestor deplasări cu funcţii şi nu cu tehnici de indexare.

În cazul programelor secvenţiale, valorile la margine vor fi calculate, în mod normal, separat; în cazul construcţiilor paralele acest lucru se poate face prin considerarea unei anumite geometrii a masivului. De exemplu, o dimensiune poate fi considerată ciclică, în care caz deplasările se fac ca rotaţii, sau planară, cînd deplasările sînt fără sfîrşit, cu transferul corespunzător al datelor la margine. În fig. 4.5 se prezintă combinaţiile ale acestor geometrii pentru un masiv bidimensional. Se pot observa suprafeţele topologice planară, cilindrică şi toroidală.

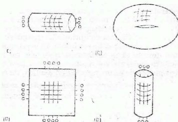


Fig. 4.5 Geometriile topologice posibile în două dimensiuni folosind geometriile planară şi ciclică pentru deplasarea masivului: (a) planară; (b) cilindrică (axa N—S); (c) cilindrică (axa E—V); (d) toroidală.

Evident sintaxa exemplului anterior limitează deplasările la o poziţie în fiecare direcţie, deoarece adăugarea unui întreg la + şi — creează o selecţie corectă din A. Limbajele paralele încorporează deplasări cu mai mult de o poziţie prin folosirea simbolului selector pentru întreaga dimensiune, caz în care A(∗ — N) deplasează A după prima dimensiune, N poziţii la dreapta.

(iii) Expresii masiv şi conformitate

Odată definite în cadrul unui limbaj masivele ca obiecte de date, este necesar să se definească reguli de utilizare a lor în expresii. Regula fundamentală priveşte modul de aplicare a operatorilor elementelor individuale ale obiectelor masiv. Cînd se combină masive cu operatori aritmetici, relaţionali sau logici, se presupune că elementele corespunzătoare ale celor două masive se combină cu acelaşi operator, ca şi cum operaţia s-ar executa simultan. Operatori unari similari se aplică tuturor elementelor unui obiect masiv. Rezultatele obţinute sînt tot obiecte masiv. Odată definite operaţiile dintre masive, se pot construi expresii ca în limbajele secvenţiale, prin folosirea regulilor acceptate privind tipul, ordinea operatorilor şi plasarea parantezelor. Cele mai importante consideraţii sînt regulile privind dimensiunea şi forma obiectelor masiv ca operanzi în cadrul expresiilor. Ele sînt fundamentale pentru orice limbaj.

Atît timp cît două obiecte nu au acelaşi număr de elemente, definiţia operaţiei elementare cu masive devine ambiguă. Această situaţie poate fi evitată prin definirea modului în care se suprapun operanzii masiv, împreună cu o valoare dată elementelor nedefinite. De exemplu, operanzii masiv pot fi aliniaţi prin primele sau ultimele lor elemente, liniar sau dimensiune cu dimensiune. Elementelor nedefinite li se pot da valoarea unitate sau

nulă, sau pot fi lăsate nedefinite etc. Evident, situația aceasta nu este de dorit.

Alternativa este de a impune perechii de operanzi dintr-o expresie să se conformeze. Această noțiune se definește aici prin aceea că operanzii trebuie să aibă același rang și același domeniu pentru dimensiunile corespunzătoare. Prin limitarea expresiilor în acest mod, este posibil să se exercite un control mai riguros asupra apariției erorilor, atît la compilare, cît și la execuție. Exemplul îl oferă produsul matricelor de la sfîrșitul acestei secțiuni. Odată definită restricția, devine necesar să se introducă construcții care să asigure posibilitatea comprimării, expandării sau modificării formei masivelor, astfel încît să fie conforme. În acest mod, sarcina de a specifica exact cum dorește să se conformeze masivele va reveni programatorului. Tehnicile de indexare vor reprezenta mecanisme utile, în acest sens ; altele pot fi asigurate fie ca operatori suplimentari, fie ca funcții ale limbajului.

(iv) Constrîngerea masivelor pentru a se obține conformitate

Se va folosi o notație funcțională pentru a ilustra unele din operațiile necesare pentru a constrînge masivele să se conformeze. Funcțiile prezentate aici vor fi necesare într-o anumită formă în orice limbaj care impune condiția de conformare a operanzilor masiv.

Funcții de reducere a rangului. S-a arătat că rangul unui obiect masiv poate fi redus prin indexare sau selecție. Un alt mod în care poate fi redus este prin folosirea repetată a unui operator binar între elementele uneia sau a mai multor dimensiuni ale masivului. Deși acest lucru se poate scrie ca o secvență de operații, nu ar oferi posibilitatea folosirii paralelismului pentru reducere. De exemplu, suma a N elemente se poate calcula în $\log_2 N$ pași în paralel (vezi § 5.2.2). Tab. 4.1 prezintă o listă a operațiilor de reducere cele mai obișnuite. Fiecare are doi parametri, masivul și dimensiunea după care se execută reducerea. Alte funcții ar trebui să permită reducerea după toate dimensiunile masivului. Funcții similare sînt date în APL ca operatori compuși unde $\mathcal{C}$/reprezintă reducerea cu operatorul binar $\mathcal{C}$. Sintaxa acestor operații compuse este descrisă de Iverson (1979).

Funcții de creștere a rangului. Adesea este necesar să se crească rangul unui obiect pentru obținerea conformității. Utilizarea cea mai frecventă este în operații între scalari și masive. De obicei, în calculul paralel se folosește termenul de emisie (broadcasting), deoarece se emite o copie a scalarului ca al doilea operand la fiecare element al masivului. Această acțiune poate fi interpretată, mai general, ca o constrîngere a scalarului, prin repetiție, la un masiv conform. Acest exemplu particular este lipsit de ambiguitate și poate fi pus în practică prin relaxarea regulii care impune conformarea operanzilor, permițînd scalarilor și masivelor să fie folosiți liber în expresii.

În cazul mai general acest lucru nu mai este adevărat. De exemplu, se pot executa operații între vectori și matrici fie prin repetarea vectorului ca o linie sau coloană, pentru a obține conformitate cu matricea. În general, există ^nC_r moduri în care un masiv „pătratic" r-dimensional poate fi constrîns la un masiv „pătratic" n-dimensional, unde

$$^nC_r = \frac{n!}{r!\,(n-r)!}$$

Tabelul 4.1 **Funcţii de reducere pentru extensii masiv ale limbajului FORTRAN. A este întreg sau real, B este logic. Trebuie să se asigure că aceste funcţii se pot aplica după toate dimensiunile, probabil prin omiterea celui del-al doilea parametru.**

Funcţia	Operaţia	Reducere
SUM (A, k)	+	$\sum_{i_k} A(i_1 \ldots, i_k, \ldots, i_l)$
PROD (A, k)	×	$\prod_{i_k} A(i_1 \ldots, i_k, \ldots, (i_l)$
AND (V, k)	∧	$\text{and}_{i_k}\, B(i_1 \ldots, i_k, \ldots i_l)$
OR(B, k)	∨	$\text{or}_{i_k}\, B(i_1, \ldots, i_k, \ldots, i_l)$
MAX (A, k)	⩾	$\max_{i_k} \{A(i_1 \ldots, i_k, \ldots, i_l)\}$
MIN (A, k)	⩽	$\min_{i_k} \{A(i_1 \ldots, i_k, \ldots, i_l)\}$

Dacă masivele au domenii diferite pentru fiecare dimensiune, numărul posibilităţilor se restrînge, sau chiar dispare. Este o idee bună de a folosi constrîngerea ca o operaţie explicită sau apel de funcţie, deoarece se poate asigura verificarea erorilor. Din nou, se poate folosi exemplul înmulţirii matricelor. Pentru orice constrîngere prin repetiţie este necesară numai o funcţie. Astfel, de exemplu,

$$\mathbf{XPND(A, k, N)}$$

este un masiv format prin repetarea lui **A**, de N ori după dimensiunea **k**. De exemplu, dacă **V** este un vector cu N elemente atunci

$$\mathbf{XPND(V, 1, N)}$$

este formată prin repetarea lui V drept linii ale matricei.

În continuare se poate realiza orice constrîngere prin folosirea repetată a acestei funcţii simple.

O construcţie pentru modificarea formei. O altă construcţie care poate fi necesară pentru obţinerea conformităţii este modificarea conceptuală a formei structurilor de date. Uneori sînt necesare operaţii între masive de dimensiuni diferite, dar cu acelaşi număr total de elemente. Algoritmul transformatei Fourier rapide (§ 5.5.2) este un exemplu bun, deoarece este mai simplu să se prelucreze un masiv liniar cu N elemente ca o variabilă masiv tri-dimensională. Un program de la sfîrşitul acestei secţiuni exemplifică acest lucru.

Cu cît se pune un accent mai mare pe forma masivelor, care trebuie să se conformeze în expresii, este mai important de definit forma obiectelor masiv trecute subprogramelor. De exemplu, un parametru şi un argument ar trebui să se conformeze numai dacă forma sau dimensiunea acestor obiecte ar fi aceeaşi. Astfel, la FORTRAN, un parametru masiv pentru un subprogram trebuie să definească submasivul principal al argumentului transmis. În acest mod se poate selecta în subprogram un subset parametrizat al domeniului fiecărei dimensiuni.

Dacă instrucțiunea DIMENSION poate defini un domeniu dinamic în cadrul subprogramului, este necesară o instrucțiune care să schimbe forma masivului în raport cu declarația inițială. Pentru claritate folosim instrucțiunea MAP, care are formatul unei instrucțiuni DIMENSION, dar care poate fi considerată executabilă, deoarece definește o restructurare a unui obiect masiv declarat. Trebuie să se presupună că această aplicație este bijectivă pentru a obține o definiție exactă.

De exemplu, dacă considerăm lista dimensiunilor $n_1, \ldots, n_p$ a obiectului masiv

$$A(n_1, \ldots, n_p)$$

atunci această listă definește forma și structura lui **A** pentru un set ordonat de $\prod_{k=1}^{p} n_k$ locații de date (care nu trebuie să fie în mod necesar contigue). Un element din această mulțime este localizat cu ajutorul unei liste index, $i, , \ldots, i_p$ care este folosită pentru generarea funcției f_D, care produce poziția elementului în mulțime:

$$f_D(i_1, \ldots, i_p) = i_1 + n_1(i_2 - 1) + \ldots + \prod_{k=1}^{p-1} n_k(i_p - 1)$$

Instrucțiunea **MAP** va redefini această funcție pe aceeași mulțime ordonată de elemente, sub forma unei liste în instrucțiunea MAP, $m_1, \ldots, m_q$

$$\text{MAP A } (m_1, \ldots, m_q)$$

Astfel se redefinește forma lui **A**, astfel că pe baza unei liste noi index, $i_1, \ldots, i_q$, se selectează un element cu funcția f_M:

$$f_M(i_1, \ldots, i_q) = i_1 + m_1\ (i_2 - 1) + \ldots + \prod_{k=1}^{q-1} m_k(i_q - 1)$$

Se poate vedea că aplicația nu este complet definită dacă nu se respectă egalitatea

$$\prod_{k=1}^{p} n_k = \prod_{k=1}^{p} m_k$$

Trebuie să observăm că f_D nu se definește pe o mulțime contiguă de locații de memorie. Motivul este că mulțimea ordonată de elemente pe care este definită f_D, poate reprezenta tot o modificare (o reducere a domeniului) a unui masiv transmis unui subprogram.

La un calculator serial, aceste funcții ar fi folosite mai degrabă pentru calculul adreselor. Totuși, la unele mașini paralele, reorganizarea datelor poate fi necesară pentru asigurarea paralelismului.

În continuare se prezintă un exemplu de utilizare a instrucțiunii MAP: modifică forma unui masiv liniar, în două dimensiuni.

```
      DIMENSION A(N)
           ⋮
      MAP A(N/2, 2)
           ⋮
         39
```

Se poate vedea că dacă N este impar, reorganizarea lui A nu este complet definită. Ultimul element al lui A nu este definit în MAP, datorită trunchierii rezultatului împărţirii a două numere întregi.

(*v*) *Indexarea expresiilor*

Prin introducerea paralelismului în limbaj, şi expresiile au devenit obiecte masiv, cu o formă dată. De aceea, este de dorit ca selecţia dintr-o expresie să se facă cu aceleaşi tehnici de indexare descrise anterior. Este probabil ca selecţia să fie folosită cu forma funcţională şi prin evitarea ulterioară a atributelor ne-necesare la variabile temporare. Sintactic, selecţia poate fi executată prin adăugarea la sfîrşitul expresiei a unei perechi de paranteze cu selectorul necesar. În acest mod, expresia este tratată exact ca un obiect masiv. Primul program este un exemplu în acest sens.

(*vi*) *Atribuirea masivului*

Într-o anumită fază, expresia masiv trebuie atribuită unei variabile masiv. Atribuirea, ca orice altă operaţie, este elementară şi, folosind aceleaşi argumente pentru evaluarea expresiei, rezultatul trebuie să fie conform variabilei masiv. Din nou se poate relaxa conformitatea pentru asigurarea unei expresii scalare la o variabilă masiv, deoarece aceasta se poate realiza fără ambiguităţi. Pentru a realiza conformitatea la faza de atribuire, se pot folosi toate mecanismele de selecţie descrise anterior, pentru actualizarea selectivă a unei variabile masiv.

Astfel, de exemplu, se poate atribui selectiv o expresie vectorială unei linii sau coloane a unei matrici cu

A(I,) = <expresie vectorială conformă>

sau

A(,J) = <expresie vectorială conformă>

O tehnică mai puternică este atribuirea selectivă cu un masiv sau expresie booleană. Aceasta poate de fapt înlocui atributele condiţionate. Uneori tehnica este denumită mascare, deoarece masivul boolean poate fi considerat ca o mască care controlează atribuirea. De exemplu, se poate genera valoarea absolută a lui A prin inversarea numai a elementelor negative :

A(A.LT.0) = — A

(*vii*) *Exemple de programare*

Folosim acum construcţiile de mai sus pentru a ilustra modul cum codul paralel poate reprezenta o interpretare concisă şi uşor de înţeles a unui algoritm. Se folosesc trei exemple : problema generală a reorganizării după o dimensiune, înmulţirea matricelor şi algoritmul pentru transformata Fourier rapidă.

Reorganizarea generală. Această problemă poate fi ilustrată limpede folosind selecţia unui cuvînt sau frază dintr-un alfabet. Astfel, fiind dat un masiv uni-dimensional de caractere, ALPHABET (27), care conţine caracterele A la Z şi blanc, dorim să selectăm un cuvînt sau

frază, WORD(N), din alfabet cu un masiv întreg INDEX(N) care indică literele corespunzătoare din alfabet.

Să ne reamintim că indexarea indirectă care se poate realiza este de timpul proecției sau reducerii rangului și nu de forma generală cerută de acest exemplu. Prin urmare, tehnica constă în expandarea alfabetului într-un obiect bi-dimensional prin N repetiții. Apoi, se poate selecta cuvîntul prin indexarea primei dimensiuni, folosind cele N elemente ale INDEX-ului. Se obține programul de mai jos. Structurile manipulate sînt ilustrate în fig. 4,6.

```
FUNCTION WORD (ALPHABET INDEX,N)
CHARACTER ALPHABET (27), WORD(N)
INTEGER INDEX(N)
C
WORD=XPND (ALPHABET, 2, N) (INDEX,)
RETURN
END
```

Înmulțirea matricelor. Al doilea exemplu este înmulțirea matricelor. Se prezintă corespondența dintre notațiile matematice și de programare. De asemenea, în acest exemplu, cele NLM înmulțiri sînt exprimate în paralel, fără pierderea generalității algoritmului.

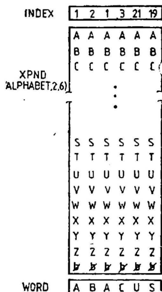


Fig. 4.6 Structurile de date manipulate în exemplul programului de aplicație generală.

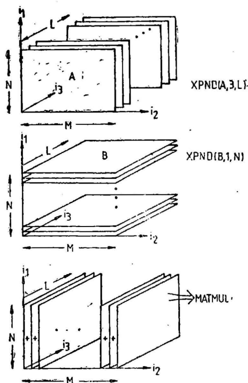


Fig. 4.7 Structurile de date manipulate în exemplul programului de înmulțire a matricelor.

Produsul se obține prin expandarea, în prima fază, a lui **A** și **B** sub forma unor obiecte tri-dimensionale (fig. 4.7). La început se repetă **A** ca primele două dimensiuni ale structurii, apoi se repetă **B** ca ultimele două dimensiuni. Se poate observa că în ambele cazuri se apelează **XPND** care produce masive conforme (de exemplu, ambele au forma (N, M, L)). Apoi, se reduce produsul acestor masive prin sumarea corespunzătoare dimensiunii din

```
      FUNCTION MATMULT (A,B,N,L,M)
      REAL A(N,M), B(M,L), MATMULT(N,L)
C
C     THE MATEMATICAL DESCRIPTION OF THE PROBLEM IS GIVEN FOR I1+1,...,N
C     AND I3 = 1,...,L, BY
C
C                         M
C                         -
C    MATMULT (I1,I3)=     \A       *B
C                         / I1,I2  I2,I3
C                         -
C                        I2=1
C
C    THE CODE IS VERY SIMILAR
     MATMULT = SUM(XPND(A,3,L)*XPND(B,1,N),2
     RETURN
     END
```

mijloc (pentru toate subspațiile **M** cu forma (N, L)), obținîndu-se rezultatul așteptat, o matrice $N \times L$, atribuită variabilei **MATMULT**. Să remarcăm că sub această formă, cînd matricele nu sînt pătratice, orice eroare în specificarea selectorilor dimensiunii indicate (1, 2 sau 3) va produce o eroare de compilare, deoarece fie înmulțirea, fie atribuirea nu va fi conformă.

Această expresie a algoritmului implică lipsa secvențierii (care este lăsată acum în sarcina compilatorului), astfel că plecînd de la acest cod se pot obține toate cele patru variante ale algoritmului (§ 5.3.1 — §5.3.4).

(iii) Transformata Fourier rapidă

În § 5.5.2 se deduc ecuațiile recursive care definesc transformata Fourier rapidă sub forma unei secvențe de transformate parțiale peste o structură bi-dimensională variabilă (ecuația (5.88)). De asemenea, distanța dintre elementele acestor ecuații poate fi considerată ca a treia dimensiune de lungime 2, care înjumătățește domeniul primei dimensiuni. Se obține o transformare simplă de la ecuația (5.88) la ecuațiile :

for sequence $l = 0,1,...,q-1$

  for all $k = 1,...,2^l$

   for all $i = 1,...,n/2^{l+1}$

$$\overset{(l+1)}{f}(i,1\,k) = \overset{(l)}{f}(i,1\,k) + \omega_{2^{l+1}}^{k-1}\overset{(l)}{f}(i,2,k)$$

$$\overset{(l+1)}{f}(i,2,k) = \overset{(l)}{f}(i,1,k) - \omega_{2^{l+1}}^{k-1}\overset{(l)}{f}(i,2,k)$$

  end

 end

repeat

Acum, exprimarea acestor ecuaţii cu ajutorul construcţiilor paralele este o transformare banală, folosind o funcţie RECUR (ca în fig. 5.13, 5.14 şi 5.16). Funcţia realizează un pas al secvenţei corespunzătoare lui 1.

```
FUNCTION RECUR(F,W,L,N)
COMPLEX F(N) , W(N) , RECUR(N)
I2L = 2**L
N2L = N/I2L
N2L1 = N2L/2
MAP F(N2L1,2,I2L), W(N2L, I2L), RECUR(N2L1, 2,I2L)
F( ,2, ) = XPND(W(1, ),1,N2L1)*F( ,2, )
RECUR( ,1, ) = F( ,1, ) + F( ,2, )
RECUR( ,2, ) = F( ,1, ) - F( ,2, )
RETURN
END
```

Acest cod, folosit cu o secvenţă de apel care atribuie RECUR la F, produce transformatele. Este folosit numai spaţiul de memorie temporar definit de funcţia ce foloseşte ca argumente masive şi produce valorile transformatei în ordine inversă. Lăsăm ca un exrciţiu cititorului să reprogrameze funcţia pe baza diagramei din fig. 5.12. pentru a se obţine rezultatele în ordinea naturală.

Trebuie din nou subliniat că această descriere a algoritmului încorporează toate cele trei scheme descrise în §5.5. Schema A se obţine dacă compilatorul secvenţiază ultima dimensiune a masivelor, iar schema B dacă compilatorul secvenţiază prima dimensiune a masivelor. Dacă, totuşi, maşina este suficient de paralelă, atunci fără a se secvenţia nici o dimensiune se produce schema paralelă (PARAFT) (§ 5.5.4). În acest mod se poate genera cu un compilator adecvat cod maşină eficient pentru un spectru întreg de calculatoare cu $n_{1/2}$ variind de la 1 la N.

4.3.2. DAP FORTRAN — o soluţie cu restricţii

DAP FORTRAN (ICL 1979 a, b) este un limbaj dezvoltat, aşa cum sugerează şi numele său, pentru ICL DAP. Acest limbaj este singurul de nivel înalt implementat pe calculatorul DAP (vezi §3.4.2) şi continuă să fie folosit şi de mini-DAP (vezi § 3.5.2). Acest limbaj, deşi constrînge masivele de date să aibă dimensiunea şi forma masivului de procesoare ţintă, are contribuţii importante la dezvoltarea ulterioară a standardului FORTRAN pentru masive de procesoare. Implementează multe din construcţiile descrise în secţiunea anterioară, deşi limitate la obiecte masiv cu dimensiunea 64 × 64 pentru DAP şi 32 × 32 pentru mini-DAP. De asemenea, limbajul este sursa mai multor funcţii intrinseci orientate-masiv propuse în cadrul standardului FORTRAN 8X.

(i) Obiecte de date

În DAP FORTRAN masivele sînt declarate în modul obişnuit, folosind instrucţiunea DIMENSION SAU TYPE. Totuşi, vectorii sînt declaraţi cu prima dimensiune elidată, iar matricele cu primele două elidate. Acestea sînt dimensiunile supuse restricţiilor, care iau la DAP mărimea de

N elemente pentru un vector şi N × N pentru o matrice. De asemenea, se pot declara mulţimi de obiecte, prin folosirea altor dimensiuni.

DIMENSION V( ), VSET(,4)
REAL M(,), MSET (,,4)

Aceste exemple definesc un singur vector **V** şi o mulţime de vectori **VSET**, urmaţi de o matrice **M** şi o mulţime de 4 matrici **MSET**. Dimensiunile care nu sînt supuse restricţiilor sînt folosite numai pentru accesul secvenţial şi corespund memoriei DAP. Detalii se găsesc în § 3.4.2. Aceste dimensiuni sînt considerate ca în standardul FORTRAN.

Deoarece dimensiunile paralele sînt limitate de caracteristicile masivului DAP, obiectele de acelaşi tip vor fi conforme. Cînd, însă, se folosesc obiecte de tipuri diferite, trebuie folosite regulile de conformare. DAP FORTRAN permite transformarea scalarilor în tipurile vectorial sau matricial şi posedă funcţii care permit trecerea vectorilor la tipul matricial. Funcţiile MATR şi MATC formează matrici prin repetarea vectorului ca linii, respectiv coloane.

(ii) Mecanisme de selecţie

Mecanismele de selecţie pentru dimensiunile supuse restricţiilor folosesc indici întregi, logici sau vectoriali, aşa cum s-a prezentat în § 3.4.2. Indexarea vectorială sau indirectă poate fi aplicată ca proiecţie peste toate liniile sau coloanele. Totuşi, această tehnică nu poate fi folosită pentru celelalte dimensiuni, deoarece ar necesita memorii indexate separate la fiecare PE, lucru care nu există.

Selecţia corespunzătoare dimensiunilor supuse restricţiilor este cu reducerea rangului, cu excepţia actualizării selective (vezi § 4.3.1). Astfel, selecţia dintr-o matrice produce fie un singur vector fie un scalar, iar selecţia dintr-un vector produce întotdeauna un scalar. Nu există nici un mecanism de selecţie după domeniu, iar selecţia logică trebuie să producă un rezultat unic. În continuare se prezintă exemple pentru aceste mecanisme de selecţie, pentru următoarele declaraţii :

REAL **M**(,), **V**( )
INTEGER I , J, **VI**( )
LOGICAL **ML** (,), **VL** ( )
M(I,) linia I din **M**,
M(,J) coloana J din **M**,
M(I, J) elementul I, J din **M**,
V(I) elementul I din **V**,
M(**VL**,) linie a lui **M**,
M(,**VL**) coloană a lui **M**,
M(**ML**) element al lui **M**,
V(**VL**) element al lui **V**,
M(**VI**,) vector ce conţine (**M**(**VI**(I), I) în elementul I
M(**VI**,) vector ce conţine **M**(I, **VI** (I)) în elementul I.

În aceste exemple, matricele sau vectorii logici trebuie să aibă numai un element cu valoarea logică. TRUE., iar vectorul întreg **VI** trebuie să aibă toate elementele sale în domeniul 1 la N. La DAP FORTRAN, indexarea se poate generaliza prin acceptarea unor expresii adecvate în locul variabilelor din exemplele de mai sus.

Se poate folosi ca operaţie de indexare şi deplasarea, prin utilizarea simbolurilor + sau — în cadrul dimensiunilor supuse restricţiilor.

V(+) deplasează **V** o poziţie la stînga
V(−) deplasează **V** o poziţie la dreapta
M(+,) deplasează **M** o poziţie la nord
M(−,) deplasează **M** o poziţie la sud
M(,+) deplasează **M** o poziţie la vest
M(,−) deplasează**M** o poziţie la est
M(+) deplasează **M** o poziţie la stînga, tratată ca un vector mare
M(−) deplasează **M** o poziţie la dreapta, tratată ca un vector mare

Deplasări cu mai mult de o poziţie se exprimă prin intermediul funcţiilor.

(*iii*) *Atribuirea masivelor*

Expresii indexate aflate în membrul stîng al unei atribuiri specifică ce selecţie din variabilă va fi actualizată. Atribuirea selectivă poate fi folosită pentru conformare; de exemplu, o expresie vectorială poate fi atribuită oricărui vector selectat dintr-o matrice. Sau, selecţia poate fi folosită la actualizarea unui domeniu de valori, cînd expresia şi membrul stîng sînt conforme. În acest caz, selecţia indică numai care elemente vor fi modificate. De exemplu, să considerăm indexarea cu o matrice logică sau un vector logic:

M(LM) = <expresie matricială>
M(VL,) = <expresie matricială>

În ambele cazuri, restricţiile plasate în membrul drept nu se mai aplică Matricea logică LM sau vectorul logic LV poate avea cel puţin un element cu valoarea logică adevărată. Astfel, LM selectează o mulţime de elemente ce urmează să fie actualizate, iar LV o mulţime de vectori linie. În ambele cazuri, elementele selectate pentru actualizare primesc valorile elementelor corespunzătoare din expresia matricială iar celelalte rămîn neschimbate.

(*iv*) *Funcţii*

Deoarece în DAP FORTRAN expresiile pot avea valori matriciale sau vectoriale, definiţia subprogramului funcţie a fost extinsă pentru a include aceste situaţii, ca şi funcţia mai uzuală cu valoare scalară. Tipul funcţiei se declară în instrucţiunea de definire a funcţiei. Astfel,

REAL MATRIX FUNCTION MATMULT

declară o funcţie MATMULT care produce o matrice reală.

În plus la extinderea noţiunii de funcţie definită de utilizator, funcţiile standard, sau interne, FORTRAN sînt făcute să fie polimorfice. Ele produc o valoare cu acelaşi tip ca argumentul lor, tipurile matricial şi vectorial fiind evaluate în paralel. Alte funcţii interne ale DAP FORTRAN-ului execută manipularea de date, deplasări, selecţii sau reduceri. Funcţiile de reducere lucrează fie asupra uneia sau a ambelor dimensiuni supuse restricţiilor, folosind operatori aritmetici, relaţionali sau logici. Iată cîteva exemple:

ALL.AND. peste linii şi coloane
ANY .OR. peste linii şi coloane

ANDROWS .AND.peste linii
ORCOLS .OR. peste coloane
SUM + peste linii şi coloane
MAXP ⩾ peste linii şi coloane (produce numai poziţia logică).

(*v*) *Exemple*

În exemplul general de mapare, codul DAP FORTRAN arată similar celui prezentat pentru construcţiile paralele generale (§4.3.2). Motivul este faptul că dimensiunea problemei corespunde masivului DAP sau dimensiunilor supuse restricţiilor în DAP FORTRAN. Toutşi, codul este bun numai pentru N ⩽ 64(32). Masivele ALPHABET şi WORD trebuie completate cu blancuri în poziţiile rămase neocupate pînă la 64 sau 32.

```
CHARACTER VECTOR FUNCTION WORD (ALPHABET, INDEX)
CHARACTER ALPHABET ( )
INTEGER INDEX ( )
WORD = (MATC (ALPHABET)) (INDEX),
RETURN
END
```

Să observăm că sintaxa impune includerea expresiei în paranteze, pentru a o indexa.

(*vi*) *Exemplul înmulţirii matriciale*

Din nou, datorită restricţiilor impuse dimensiunilor, codul ce urmează înmulţeşte matrici mai mici sau egale cu matricea DAP. Varianta de algoritm folosită este produsul extern, care are paralelism N^2 (§ 5.3.3).

```
REAL MATRIX FUNCTION MATMULT (A, B, N)
REAL A (,), B (,)
INTEGER K, N
MATMULT = 0
DO 10 K = 1, N
10 MATMULT = MATMULT + MATC (A(, K)) * MATR (B (K,))
RETURN
END
```

În DAP FORTRAN poate fi implementată o variantă a expresiei generale a înmulţirii matriciale, pentru matrici cu dimensiunea 16 × 16, 16 fiind rădăcina cubică din 64^2 :

MATMULT (ROW(I3). AND. COL (J3)) = SUM — 4 — SQUARE(
MATC — 4 — SQUARE (A, I1, J1)*
MATR — 4 — SQUARE (B, I2, J2), I3, J3)

unde SUM — 4 — SQUARE, MATC — 4 — SQUARE şi MATR — 4 — SQUARE sînt funcţii definite de utilizator (Jesshope şi Craigie 1980), care simulează un DAP 16^3, prin folosirea unei suprapuneri simetrice peste masivul DAP 64^2 (Jesshope 1980c).

4.3.3. FORTRAN 8X — noul standard FORTRAN

Deşi în cadrul limbajului DAP FORTRAN, obiectele de date paralele corespund arhitecturii ţintă, alte limbaje propuse aproximativ în aceeaşi

perioadă de timp au adoptat soluţii lipsite de restricţii pentru introducerea paralelismului structural în FORTRAN. Două astfel de exemple sînt compilatorul BSP FORTRAN, care combină beneficiile vectorizării cu cele oferite de construcţiile masiv (Burroughs 1977e,Austin 1979) şi VECTRAN un limbaj experimental dezvoltat de IBM, care se bazează pe FORTRAN şi este prezentat în lucrările lui Paul şi Wilson (1975, 1978).

Evident, dezvoltarea separată a mai multor limbaje pentru arhitecturi de masive sau vectoriale nu este o situaţie ideală. Din acest motiv, Comitetul ANSI X3J3 a decis luarea în considerare a extensiilor masiv ca o caracteristică ce urmează să fie adăugată la noul standard FORTRAN. Această posibilitate a fost luată în considerare pentru prima dată de comitetul ANSI care a produs standardul FORTRAN 77, dar abandonată cu recomandarea de a fi inclusă în standardul următor. Multe din aceste caracteristici se bazează pe construcţii întîlnite în limbaje ca DAP FORTRAN, BSP FORTRAN şi VECTRAN. Deliberările au început cu aproape o decadă în urmă, iar Comitetul X3J3, în momentul redactării acestei lucrări a votat specificaţiile următorului standard FORTRAN. Acesta este denumit FORTRAN 8X, numele reflectînd acordul final ANSI, în anul 198X. Secvenţa (66, 77, 88) pare din ce în ce mai improbabilă, deoarece standardul va fi aprobat numai după ce Comitetul va analiza răspunsurile oferite prin recenzii publice. Comitetul are dificultăţi în asigurarea unor specificaţii de proiect ce ar urma să fie discutate public, deoarece mulţi delegaţi cred că limbajul a devenit prea bogat.

Noile caracteristici ale standardului propus care ne interesează aici, sînt cele referitoare la masive şi prelucrarea lor (paralelismul procesului sau task-ului nu este o noutate a limbajului). Cele prezentate aici se bazează pe informaţiile obţinute din documentul de lucru ANSI X3J3 (X3J3/S8, versiunea 95) data iunie 1985, de aceea nu trebuie considerate ca imuabile. Ele pot fi schimbate, înlocuite sau introduse înainte de a fi acceptate ca standard. Cu această menţiune, vom adăuga că aceste caracteristici se menţin de mai mult timp.

Extensiile masiv ale noului limbaj urmează îndeaproape propunerile expuse în § 4.3.1. Totuşi , s-a mai definit un tip de date, ca o consecinţă a prelucrărilor de masive. Acesta este tipul BIT, care are două valori, „1" şi „0" şi care împreună cu operaţiile binare reprezintă sistemul cu două valori al matematicii booleene.Acest tip a fost necesar pentru a permite folosirea variabilelor de mascare booleene care permit sau întîrzie operaţiile cu masive.

(*i*) *Masive ca obiecte elementare de date*

Obiectele masiv sînt de formă dreptunghiulară, cu un index pentru fiecare dimensiune (pînă la 7) a structurii. Mărimea fiecărei dimensiuni este denumită *întinderea* (extent) acelei dimensiuni, iar numărul dimensiunilor, *rangul* (rank) masivului. Un scalar are rangul 0. Mărimea masivului este numărul total de elemente sau produsul întinderilor. Un masiv poate avea mărimea 0. *Forma* (shape) unui masiv este definită de rang şi întinderea dimensiunilor. Odată declarat, rangul masivului este constant, dar întinderile pot să nu fie constante. Întinderile pot varia pentru argumente nedefinite, masive incluse în proceduri locale, detalii fiind prezentate în conti-

nuare. Rangul unui masiv se poate schimba, în timp ce mărimea rămîne constantă, prin intermediul unui subprogram cu o interfaţă parametru/argument nedefinit.

Se spune că două masive sînt conforme în limbajul nou dacă au aceeaşi formă, cu excepţia scalarilor, care sînt conformi cu orice masiv. Aşa cum s-a prezentat în § 4.3.1, orice operaţie definită pentru scalari poate fi aplicată masivelor conforme şi acestea se execută ca şi cum operaţia scalară s-ar aplica fiecărui element al masivelor şi într-un astfel de mod încît se poate presupune că toate operaţiile se execută simultan.

(ii) Masive cu forma explicită

Un masiv cu forma explicită este definit cu valori explicite pentru limitele dimensiunii masivului, o valoare inferioară optimă şi una super oară pentru fiecare dimensiune a masivului. Dacă se folosesc variabile pentru definirea acestor limite, atunci masivele trebuie să fie argumente nedefinite, sau variabile locale într-o procedură. Iată masive cu forma explicită :

A(2 : 10,5 : 30), B(10, 10, 10), C(N — 1)

Aici A este un masiv bi-dimensional, cu intinderea de la 2 la 10 în prima dimensiune şi de la 5 la 30 pentru a doua dimensiune ; B este un masiv cu trei dimensiuni cu întinderea de la 1 la 10 pentru toate dimensiunile. Dacă N ar fi o variabilă, atunci masivul C ar trebui să fie un argument nedefinit sau variabilă locală într-o procedură.

(iii) Masive cu forma primită

Un masiv are forma primită dacă moşteneşte forma de la un argument actual procedurii. Un astfel de masiv apare numai într-o procedură, unde este declarat ca argument nedefinit. Va lua forma argumentului actual cînd este referită procedura. Argumentul poate fi declarat într-o instrucţiune ARRAY, care conferă atribute de masiv unei variabile. În acest caz argumentul nedefinit, ca şi celelalte entităţi declarate în aceeaşi instrucţiune ARRAY vor primi forma argumentului actual, care devine asociat cu argumentul nedefinit. În acest mod, un număr de obiecte masiv vor primi forma unui parametru masiv.

(iv) Masive cu forma alocată

Un masiv cu forma alocată are tipul, numele şi rangul declarate, dar forma şi limitele sînt determinate în faza de execuţie cînd se alocă spaţiu cu o instrucţiune ALLOCATE. Iată exemple :

A(:, :), B(:, :, :), C(:)

Rangul masivului este dat prin numărul de coloane, astfel că acest masiv are acelaşi rang cu masivele declarate explicit anterior. Mărimea şi forma unor astfel de masive sînt predefinite pînă la alocarea lor. Pînă ce nu sînt alocate masivele, nu se poate face nici o referinţă la ele sau elementele lor.

(*v*) *Secţiuni masiv*

Ca şi în cazul construcţiilor definite în §4.3.1, o referinţă la un masiv fără indici implică utilizarea întregului masiv. Dacă se execută anumite calcule, ele se vor face cu toate elementele şi în orice ordine. Există excepţii cînd este necesară ordinea lexicală, ca în cazul instrucţiunilor de I/E. Se pot selecta elemente din masiv, pe baza unei liste complete a indicilor, în modul obişnuit. Numărul necesar de indici este dat de rangul masivului. Fiecare variabilă indice reduce rangul secţiunii selectate cu 1, astfel încît dacă sînt furnizaţi toţi indicii, rangul secţiunii selectate va fi 0, c-ea ce corespunde unui scalar.

Altă posibilitate este de a folosi domenii ale indicilor în poziţiile corespunzătoare. Prin urmare, secţiunea se defineşte (ca la masivul părinte) prin produsul cartezian al acestor domenii de indici. Submulţimea se selectează din domeniul indicelui prin intermediul unui selector, care ia una din următoarele două forme : un triplet ce defineşte baza, întinderea şi pasul, ceea ce produce o secvenţă monotonă, sau un masiv întreg de rang 1. Folosind tripletul (separat prin :)

B(1 : 10, 1, 1) şi B(1 : 10 : 2, 1 : 10, 1 : 10)

sînt ambele secţiuni valide din masivul cu forma explicită definit anterior. Primul este un vector cu 10 elemente, definit cu indici în ultimele două dimensiuni, iar al doilea, un masiv tri-dimensional format cu indici impari din prima dimensiune.Exact aceleaşi secţiuni s-ar fi putut defini folosind un masiv cu o dimensiune în poziţia primului indice,

B(IV, 1,1) şi B(IV, 1 : 10, 1 : 10)

unde, în primul caz, IV = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], iar în al doilea IV = [1, 3, 5, 7, 9]. În general, numărul indicilor şi al selectorilor trebuie să corespundă rangului masivului părinte. Selectorului secţiunii vectoriale poate conţine orice secvenţă de indici care se află în interiorul domeniului corespunzător indicelui respectiv.

Aşa cum s-a menţionat în § 4.3.1, lista indicilor poate defini o corespondenţă unul-la-mai mulţi, de la părinte la secţiunea masiv, ceea ce poate conduce la rezultate nedeterministe dacă sînt acceptate atribuiri pentru astfel de secţiuni masiv. Definiţia 8X interzice astfel de atribuiri, ca şi cele prin pereche parametru/argument nedefinit, dacă nu sînt deterministe.

(*vi*) *Atributul ALIAS şi instrucţiunea IDENTIFY*

Cînd un masiv nu este masiv ? Răspunsul este că atunci cînd are atributul ALIAS. Un nume de masiv declarat cu acest atribut nu este un obiect masiv pînă ce nu a fost asociat cu un obiect actual prin apariţia într-o instrucţiune IDENTIFY. Această instrucţiune are aspectul unei instrucţiuni COMMON executată dinamic, deoarece supraîncarcă o zonă de memorie cu un număr de masive diferite, care pot avea forme diferite. Spre deosebire de instrucţiunea COMMON, atribuirea unei zone de memorie unui masiv anume (ALIASed) se poate realiza cu o instrucţiune executa-

bilă, care poate conţine variabile care definesc corespondenţa masivului atribuit (ALIASed) peste obiectul masiv actual.

Probabil, instrucţiunea IDENTITY este una din cele mai controversate noi caracteristici privind posibilităţile de lucru cu masivele. Reprezintă un mijloc pentru asigurarea unei corespondenţe dinamice de adrese de la masivul alias la masivul părinte, deşi este necesar ca această corespondenţă să se realizeze liniar. Seamănă cu o secţiune de masiv, doar că asigură un domeniu mai mare de submulţimi ale masivului părinte. Un exemplu simplu va ilustra utilizarea acestei instrucţiuni.

IDENTIFY(DIAG(I) = ARRAY(I, I, I = 1 : N)

În acest exemplu, masivul DIAG este de rang 1 şi reprezintă masivul ALIAS; a fost asociat cu spaţiul de memorie al masivului părinte ARRAY, astfel încît elementele lui DIAG să corespundă diagonalei lui ARRAY. În general, corespondenţa cu masivul părinte poate fi orice combinaţie liniară a indicilor masivului alias. Trebuie să existe un specificator de domeniu pentru indice care defineşte domeniul peste care este definită corespondenţa. Prin urmare, se pot specifica cu o instrucţiune, o transformare de indice liniar şi un domeniu dinamic.

Un alt exemplu, prezentat în continuare, redefineşte primele 100 elemente ale unui masiv cu rang 1, VECTOR, pentru a crea un masiv alias cu rangul 2, ARRAY. Cei doi indici definesc întinderea celor două dimensiuni ale masivului ARRAY, iar corespondenţa este definită de combinaţia liniară a acestora:

IDENTIFY(ARRAY(I, J) = VECTOR(I + 10 * (J − 1)),
I = 1 : 10, J = 1 : 10)

Orice referinţă sau atribuire la masivul alias va modifica elementele masivului gazdă, cum este specificat de corespondenţa indicilor. Prin urmare, în exemplul nostru, o atribuire la DIAG va modifica primele N locaţii ale diagonalei principale din ARRAY.

Odată identificat (IDENTIFYed), un masiv alias poate fi tratat ca orice alt obiect masiv, chiar secţionat, transferat ca argument unui subprogram, sau ca obiect părinte la altă instrucţiune IDENTIFY. Utilizarea nedeterministă a procedurii ALIASing de la mai mulţi-la-unul nu este permisă, deoarece există restricţii similare celor aplicate selectorilor vectoriali.

Se poate vedea, prin urmare, că această posibilitate este foarte puternică pentru selecţia submasivelor. Totuşi, este probabil dificil de implementat pe multe masive de procesoare, cum este ICL DAP, deoarece implică manipularea în faza de execuţie a unui spaţiu global de adrese. Este mult mai indicată pentru supercalculatoarele vectoriale, unde accesul la memorie este secvenţial şi definit cu un pas constant, care asigură translarea liniară a indicelui. Ar putea fi implementat la un masiv de procesoare prin folosirea actualizării selective aplicată structurii părinte, sau dacă aceasta este prea ineficientă ca o atribuire la un masiv alias, cu operaţii ulterioare de unificare cu masivul părinte, deoarece pentru faptul că este o construcţie specifică execuţiei, este destul de probabil că vor fi necesare comunicaţii prin comutare de pachete.

(*vii*) *Funcții intrinseci și proceduri*

Deoarece semnatica noului limbaj permite manipularea obiectelor masiv ca cetățeni de prima clasă, este permisă folosirea lor ca parametri în toate funcțiile intrinseci existente în FORTRAN. Toate aceste funcții sînt polimorfice, în același mod în care operatorii sînt supraîncărcați. Astfel, SIN(A) ar produce ca rezultat un obiect care se conformează argumentului său. Prin urmare, dacă A este o matrice cu o mărime și formă date, funcția va returna o altă matrice, de aceeași mărime și formă, dar cu elementele egale cu valorile obținute prin aplicarea funcției SIN elementelor argument.

Propunerea FORTRAN 8X are multe funcții intrinseci noi, adăugate pentru a facilita introducerea extensiilor masiv în limbaj. De exemplu, există funcții enquiry care furnizează domeniul, rangul sau mărimea masivului. Mai există funcții de manipulare a masivelor, ca MERGE, SPREAD, REPLICATE, RESHAPE, PACK și UNPACK. Cele mai multe din aceste funcții au operatori echivalenți în limbajul APL (Iverson 1962).

Această secțiune nu poate reprezenta o descriere completă a noului standard FORTRAN, pentru care în momentul redactării lucrării, nu exista nici o propunere de proiect. Se încearcă crearea unei imagini despre construcțiile interesante pentru utilizatorii de calculatoare paralele vectoriale și masive. Cititorul interesat poate consulta documentul original (Campbell 1987), sau cele ulterioare, ce pot fi obținute de la secretarul Comitetului ANSI X3J3. Mai există o lucrare a doi delegați britanici în Comitetul FORTRAN X3J3 (Reid și Wilson 1985) care prezintă exemple de exploatare a posibilităților limbajului FORTRAN 8X, ca și cartea *FORTRAN 8X Explained* de Metcalf și Reid (1987).

4.3.4 CMLISP

Spre deosebire de limbajele descrise anterior, care se bazează pe structura de masiv și FORTRAN, această secțiune descrie un limbaj care se bazează pe structura de date listă și limbajul common LISP. În continuare se prezintă o introducere în LISP (pentru consecvență), iar cititorul care dorește detalii poate consulta una din cărțile numeroase despre acest limbaj (Winston și Horn 1981, Touretsky 1974).

În LISP, obiectele centrale sînt listele; chiar și funcțiile care operează asupra lor, ca și definițiile lor sînt tot liste. Funcțiile definite în LISP pot avea ca argumente alte funcții și produc funcții rezultat. Acestea sînt cunoscute ca funcții de ordin mai înalt, iar folosirea lor asigură un mediu de programare foarte puternic și expresiv. Operatorul de reducere „/" din APL este o funcție de ordin mai înalt; ca argumente are un alt operator și un masiv, aplicînd operatorul fiecărui element din masiv. De exemplu, în APL / + A va suma elementele lui A, iar /*A va forma produsul elementelor lui A.

În LISP lista este reprezentată de o secvență de articole, separate prin spațiu și incluse între paranteze. De exemplu,

item — este un atom;
(item 1 item 2) — este o listă cu doi atomi;

((item1 item2)) — este o listă cu un element, care la rîndul lui este o listă cu doi atomi;

((item1 item2) item3) — este o listă cu două elemente, unul este o listă, iar celălalt un atom.

Operațiile elementare definite în LISP realizează construirea și dizolvarea obiectelor listă, de exemplu:

CAR — produce primul atom al listei;
CDR — produce ultimul element al listei;
CONS — produce o listă pe baza unui atom și a unei liste;
LIST — produce o listă pe baza unor atomi.

CMLISP (Hillis 1985) a fost proiectat ca mediu de programare pentru connection machine și se bazează pe common LISP, care este folosit la M.I.T. de multă vreme. Este proiectat să realizeze operațiile paralele ale calculatorului SIMD connection machine. Totuși, datorită expresivității limbajului LISP este posibil să se definească în CMLISP operații de tipul MIMD. CMLISP reflectă modul de conducere a calculatorului gazdă și microcontrolerul de la connection machine, în timp ce permite exprimarea operațiilor cu structuri paralele de date. Connection machine LISP este pentru common LISP ceea ce FORTRAN 8X este pentru FORTRAN 77. Limbajul este prezentat complet în lucrarea lui Hillis și Steele (1985).

Cele trei artifacte adăugate limbajului common LISP pentru a produce CMLISP sînt xectorul, care este o expresie de paralelism; notația alfa, care este o funcție de ordin mai înalt ce exprimă paralelismul operației peste un xector; și reducerea beta, care este o funcție de ordin mai înalt care exprimă operația de reducere. Reducerea beta poate fi echivalată operatorului APL\, prezentat anterior.

(i) *Xectorul*

În CMLISP, structura paralelă de date este denumită xector, care în sens general corespunde la o mulțime de procesoare sau procesoare virtuale și valorile corespunzătoare lor. Este un obiect paralel de date ce poate fi prelucrat, furnizînd rezultatele, element după element. În acest sens amintește de masivul definit în FORTRAN 8X. CMLISP posedă operații paralele pentru crearea, combinarea, modificarea și reducerea xectorilor. Spre deosebire de DAP FORTRAN, xectorul CMLISP nu depinde de hardware; mărimea lui sau domeniul nu sînt limitate, dar pot fi definite de utilizator. Într-adevăr mărimea xectorului poate varia dinamic.

O altă diferență între FORTRAN 8X și CMLISP intervine datorită naturii structurilor de date folosite. În FORTRAN, structura de date este masivul, iar un masiv FORTRAN 8X este, în esență, o mulțime paralelă de indici și valori. Deoarece structura este implicit de formă dreptunghiulară, indicii nu sînt importanți. Manipularea acestor indici se exprimă în limbaj ca o mulțime de operatori de deplasare a datelor, care deplasează masivul într-o direcție și pe o anumită distanță. În CMLISP, xectorul constă tot dintr-o pereche index/valoare, ambele fiind obiecte LISP. Mai mult, deoarece obiectele sînt liste, adesea se va plasa pe mulțimea index un cod semnificativ. Cu alte cuvinte, xectorul reprezintă o funcție între obiecte LISP. Domeniul de definiție și de valori sînt mulțimi de obiecte LISP, iar aplicația asociază un singur obiect din domeniul de valori la fiecare obiect

din domeniul de definiţie. Fiecare obiect din domeniul de definiţie este un index şi are un obiect corespunzător în domeniul de valori, care este valoarea sa.

Implementare este astfel realizată încît se presupune că fiecare element a unui xector este memorat într-un procesor separat unde indexul este numele procesor, o adresă stocată în memoria maşinii gazdă, iar valoarea este cea memorată de acel procesor. Hillis (1985) a propus o notaţie pentru reprezentare xectorilor :

{John → Mary Paul → Joan Chris → Sue}

Aici mulţimea de simboluri John, Paul, Chris este aplicată peste mulţimea de simboluri Mary, Joan, Sue. Această notaţie reflectă interpretarea xectorului ca o funcţie. Un caz special pentru xector este mulţimea, unde o mulţime de simboluri este aplicată peste ea însăşi ;

{red → red white → white blue → blue} care este echivalent cu

{red white blue}

Un alt caz special este cel în care domeniul de valori al xectorului constă dintr-o secvenţă de întregi, începînd cu zero. Aceasta este un masiv FORTRAN 8X cu rangul unu, ce corespunde la o mulţime ordonată de valori. Notaţia alternativă sugerată de Hillis reflectă acest lucru :

{0 → 5 1 → 2 3 → 10}

care este echivalent cu

[5 2 10]

Ultimul caz special este xectorul constant care transformă fiecare valoare index posibilă într-o valoare constantă. De exemplu, xectorul care transformă toate valorile în numărul 100 este reprezentat de :

{ → 100}

Xectorii pot fi manipulaţi în LISP la fel cu alte obiecte LISP. De exemplu, un xector poate fi atribuit unei variabile folosind funcţia SETQ. De exemplu,

(SETQ Wife-of' {John → Mary Paul → Joan Chris → Sue}).

Această atribuire stabileşte valoarea simbolului 'Wife-of' xectorului definit mai sus. Simbolul ' semnifică faptul că articolul ce urmează este un atom. Odată atribuit un xector, simbolul poate fi folosit în alte funcţii ; de exemplu, pentru a referi o valoare

{XREF Wife-of'John)

se evaluează cu Mary. În mod similar, XSET va schimba o valoare pentru un index dat, iar XMOD va aduna a altă pereche de valori, dacă nu există deja. Mai există funcţii pentru conversia între xectori şi obiecte LISP obişnuite. De exemplu,

(LIST-TO-VECTOR'(5 2 10))

se evaluează cu xectorul

[5 2 10]

(*ii*) *Notația alfa*

În CMLISP, notația alfa este un mijloc mai formal de exprimare a operațiilor paralele decît cele întîlnite în alte limbaje descrise aici, de exemplu FORTRAN 8X. Strict vorbind, notația alfa este operatorul de emisie; creează un xector constant cu argumentul respectiv, de exemplu, scrie aceeași valoare la fiecare procesor. De exemplu,

$$\alpha 100 \Rightarrow \{ \rightarrow 100\}$$

$$\alpha(* \; 2 \; 3) \Rightarrow \{ \rightarrow 6\}$$

Mai interesant

$$\alpha + \rightarrow \pm \{ \rightarrow + \}$$

care este un xector de funcții +. Cînd se aplică acest obiect la doi xectori, efectul este execuția unei compoziții element cu element a valorilor xectorilor. În FORTRAN 8X, această notație este implicită, deoarece operatorii au fost supraîncărcați, pentru a reprezenta atît operații scalare cît și paralele, funcția de context. Astfel,

$$(\alpha + '\{a \rightarrow 1 \;\; b \rightarrow 3\}'\{a \rightarrow 2 \;\; b \rightarrow 4\}) \Rightarrow \{a \rightarrow 3 \;\; b \rightarrow 7\}$$

și

$$(\alpha \; \text{CONS}' \; \{a \rightarrow 1 \; b \rightarrow 3\}' \; \{a \rightarrow 2 \; b \rightarrow 4\}) \Rightarrow \{a \rightarrow \; (1.2) \; b \rightarrow (3.4)\}$$

Pentru a folosi proprietățile algebrice ale notației alfa, Hillis a introdus operatorul invers, care anulează efectul aplicării lui alfa. Este folositor dacă alfa este un factor de mai multe elemente ale unei expresii. De exemplu,

$$\alpha(+ \text{ab}) \equiv (\alpha + \alpha\text{a}\alpha\text{b})$$

dar dacă a sau b se evaluează ca xector, utilizarea lui '·' va anula aplicarea lui alfa. Astfel, dacă a ar fi fost un xector,

$$\alpha(+ \cdot \text{ab}) \equiv (\alpha + \text{a}\alpha\text{b})$$

Astfel, α este un operator paralel și ·inversul său, sau un mod de a marca obiecte deja paralele, într-o expresie evaluată în paralel.

(*iii*) *Reducerea beta*

Operatorul beta este similar operatorului de reducere APL. Cînd se aplică unei expresii ce implică un operator binar și un singur xector, are efectul modificării operatorului la unul care realizează o operație de reducere peste xector prin aplicarea operației fiecărei valori a xectorului. De exemplu,

$$(\beta + '\{a \rightarrow 1 \; b \rightarrow 3\}) \Rightarrow 4$$

Utilizarea combinată a lui alfa și a acestei notații beta în CMLISP asigură un instrument foarte puternic pentru construirea tuturor tipurilor

de funcţii. În continuare se prezintă un număr de definiţii de funcţii care folosesc această combinaţie de operatori. În aceste exemple alfa asigură emisia şi permite toate tipurile de operaţii asociative, iar beta asigură reducerea, o operaţie puternică de manipulare a mulţimilor.

(DEFUN All — Same (x, y) (β AND (α = xy))),

care se poate citi : defineşte o funcţie denumită 'All-Same', cu paramtrii x şi y (xectori), definită ca reducerea beta prin folosirea operaţiei AND a xectorului rezultat al operaţiei de egalitate între parametrii x şi y. Numai dacă toate valorile xectorilor x şi y sînt egale, va returna funcţia valoarea TRUE. O funcţie similară, dar care foloseşte operatorul logic OR se defineşte în continuare. Aceasta va detecta o egalitate între oricare două valori cu acelaşi index.

(DEFUN One—Same (xy) (βOR(α = xy)))

Următorul exemplu ilustrează folosirea operatorului punct. De fapt definiţia diferă numai în utilizarea simbolului, aplicat unuia din parametri şi factorizarea operatorului α. Funcţia Is—In are o comportare destul de diferită. Detectează dacă articolul y este în mulţimea de valori definită de xectorul x. Acum se defineşte egalitatea între y şi valorile lui x. Se folosesc alfa şi punctul pentru a realiza o expresie xector consistentă. Este o combinaţie de operatori asociativi şi de reducere :

(DEFUN Is—In(xy) (β AND α (= · xy)))

Un ultim exemplu defineşte lungimea xectorului, prin sumarea unei valori de unu la fiecare index al parametrului xector. Aceasta se realizează prin definirea unei funcţii 'Second—One', care returnează a-doilea parametru. Folosirea acestei funcţii într-o expresie alfa, folosind parametrul xector cu punct ca prim parametru al funcţiei, produce efectiv un xector de unu cu aceeaşi mărime ca x :

(DEFUN Xector—Length(x) (β + α(Second—One · x1)))
(DEFUN Second—One (xy) (y))

Utilizarea lui beta ca în exemplele de mai sus reprezintă un caz particular pentru o funcţie mai generală. În forma cea mai generală, β ia doi xectori ca argumente. Inapoiază un al treilea xector, creat din valorile primului xector şi indicii celui de-al doilea. Iată un exemplu :

(β'[1 4 9]'[A B C]) ⇒ {A → 1 B → 4 C → 9}

De fapt ,aceasta se poate interpreta ca o operaţie de transfer, deoarece transmite valorile primului xector indicilor specificaţi de al doilea xector. De exemplu, într-o reţea de transfer a pachetelor, operaţia executată de fiecare procesor ar fi emisia unui pachet în reţea, a cărui dată ar fi valoarea sa din primul xector şi a cărui adresă ar fi valoarea celui de-al doilea xector. Desigur, xectorul rezultat, definit prin sosirea acestor mesaje este nedefinit dacă al doilea xector conţine valori multiplicate. În CMLISP această condiţie reprezintă o eroare.

Totuşi, dacă este folosit alături de alt operator, se specifică o operaţie de reducere ce urmează a fi executată pentru toate conflictele de indecşi (adrese). Astfel, în timp ce

$$(\beta'[1\ 4\ 9\ 16]'[A\ B\ A\ B]) \Rightarrow \text{eroare}$$

se găseşte că

$$(\beta + '\ [1\ 4\ 9\ 16]'[A\ B\ A\ B]) \Rightarrow \{A \rightarrow 10,\ B \rightarrow 20\}$$

Cazul particular descris, cu numai un xector ca argument, presupune că al doilea argument este un xector constant, astfel că toate valorile primului xector sînt reduse de operator. Implementarea va stabili care index, sau număr de proces este supus reducerii. Într-o maşină SIMD, evident acesta este secvenţiatorul de control.

4.4 Paralelismul procesului

4.4.1 Introducere

Aşa cum s-a menţionat în secţiunea precedentă, există două tehnici pentru exprimarea explicită a paralelismului; paralelismul structural şi paralelismul procesului. De asemenea, secţiunea anterioară a prezentat diverse implementări ale primei variante, paralelismul structural. Totuşi CMLISP şi chiar APL au mecanisme care tratează operatorii ca valori elementare ale structurii paralele de date. Desigur, am putea imagina programe întregi componente ale unei structuri paralele, în care caz avem o descriere a paralelismului procesului. Pentru a ne reaminti, diferenţa este una de granularitate, deoarece paralelismul structural a fost definit ca granularitatea unei singure operaţii peste fiecare element al structurii de date. Totuşi, paralelismul procesului solicită secvenţe distribuite de operaţii.

Modelul de calcul folosit ca şi maniera distribuirii sarcinii de lucru în sistem sînt foarte diferite între aceste două alternative. Aşa cum s-a arătat în secţiunea precedentă, cineva poate considera în cazul paralelismului structural procesoarele asociate cîte unul pentru fiecare element al structurii de date, datele activate fiind repartizate procesoarelor disponibile pentru a echilibra încărcarea. Astfel, distribuirea încărcării se realizează prin redistribuirea elementelor structurii de date. În paralelismul procesului procesul este virtualizat, iar sarcina se echilibrează prin distribuirea proceselor la procesoare. Unitatea de distribuţie este codul şi, desigur, datele şi starea asociată. În acest caz virtualizarea implică menţinerea unui număr de fluxuri de instrucţiuni pe un singur procesor.

Deoarece fiecare proces implică un overhead considerabil în termenii stabilirii unui flux de instrucţiuni, există o penalitate la implementarea unui nivel înalt de concurenţă, folosind metodologia paralelismului procesului. Într-o oarecare măsură, transputerul şi OCCAM-ul este primul sistem care reduce acest overhead la dimensiuni rezonabile. Penalitatea plătită este natura statică a procesului, stabilit în faza de compilare. Crearea şi distrugerea dinamică a proceselor în timp ce este mai flexibilă, va

implica cu necesitate un overhead mai mare pentru concurență și de aceea va solicita pentru o eficientță mai bună, o granularitate mai mare. Aceasta este una din problemele importante abordate de cercetători în sistemele declarative — controlul mărimii granularității proceselor create.

Din punct de vedere istoric, paralelismul procesului a evoluat prin exploatarea partajării nedeterministe a resurselor unui singur procesor și nu din nevoia de a exploata hardware concurent. Această partajare a permis unității centrale, atunci singura resursă mai costisitoare a sistemului, să fie utilizată eficient; deoarece cînd un program are de așteptat încheierea unei operații lente de intrare sau ieșire, ciclurile CPU „pierdute" ar putea fi folosite de un alt program.

Pentru un limbaj definit pe baza conceptului de proces sînt necesare patru lucruri fundamentale : o metodă de inițiere și încheiere a task-urilor concurente (de exemplu procedurile fork și join în UNIX) ; mijloacele de comunicare între task-urile concurente, implementate fie ca un sistem de transfer de mesaje, fie printr-o memorie comună ; mijloacele de sincronizare a task-urilor, astfel încît să partajeze o referință de timp comună, deși vagă ; și, în sfîrșit, mijloace de determinare a alegerii sau metode nedeterministe. La multe limbaje, cu o singură construcție se poate rezolva mai mult de o astfel de cerință. De exemplu, alegerea memoriei partajate asigură atît mediu de comunicație, cît și nedeterminism, deoarece permite ca informațiile scrise de un task să fie citite de altul. Mai mult, dacă mai mult de un task are acces în scriere la o locație, atunci un task care citește un poate ști, fără informații suplimentare, care a fost task-ul care a scris.

O altă diferență între aceste două tehnici este că în timp ce utilizarea paralelismului structural produce un cod foarte concis, care este mai ușor de înțeles și depanat decît codul secvențial, utilizarea tehnicii paralelismului procesului creează capcane pentru programatorii neexperimentați, cum ar fi : blocarea cînd două sau mai multe procese așteaptă un eveniment sau comunicație de la unul din celelalte. Situația cea mai simplă este cînd două procese doresc să comunice între ele dar sînt ambele programate să citească de la celălalt înainte de a scrie ; ambele vor aștepta la infinit, sau pentru declanșarea unui time-out. Ruperea simetriei programate conduce la eliminarea blocărilor.

Programarea este dificilă în cadrul acestui model datorită asincronismului fundamental sau nedeterminismului global. Să presupunem că avem n fluxuri de instrucțiuni și că este posibil să stabilim o referință de timp pentru orice instrucțiune în sistem. Să considerăm că starea următoare a sistemului se obține după execuția unei instrucțiuni date. În absența oricărei sincronizări, există n alegeri posibile pentru starea următoare a sistemului, după care n^2 etc. Această creștere exponențială face depanarea foarte dificilă in cazul limbajelor ce se bazează pe procese. O situație clasică este modul în care o eroare dispare misterios cînd la programul ce funcționează defectuos se adaugă cod de urmărire a comportării sistemului. Motivul este, desigur, alterarea evoluției în timp a sistemului.

Aceasta este o secțiune mică dintr-o carte care acoperă toate aspectele paralelismului, de aceea ne limităm aici la o discuție a limbajului ce se bazează pe procese, OCCAM. Motivul alegerii acestui limbaj este asocierea sa

cu transputerul. Într-adevăr, limbajul OCCAM este foarte strîns legat de hardware-ul transputerului, de aceea recomandăm ca această parte să fie citită împreună cu § 3.5.4. unde se descrie transputerul.

Pentru mai multe detalii despre alte limbaje bazate pe procese și o discuție mai generală a aspectelor teoretice recomandăm o excelentă introducere în acest domeniu (Ben-Ari 1982) și cartea *Communicating Sequential Processes* de Hoare (1986).

4.4.2. OCCAM — o alternativă minimală

Una din schemele cele mai elegante propuse pentru sincronizarea proceselor este cea implementată la comunicația dintre procesele secvențiale (CSP) (Hoare 1978, 1986), prin comunicații punct-la—punt, fără bufere. INMOS a ales CSP ca bază a limbajului transputerului. Acest limbaj denumit OCCAM (May și Taylor 1984, INMOS 1984, Jones 1985, INMOS 1986, Pountain 1986a) este proiectat să accepte concurența hardware explicită. Prin urmare, reflectă concurența specifică transputerului, sau dacă ne exprimăm altfel, transputerele sînt proiectate să implementeze foare eficient limbajul de programare OCCAM. Deși pe transputer se pot implementa eficient cele mai multe limbaje moderne, concurența sistemelor cu transputere poate fi exploatată numai prin intermediul limbajului OCCAM.

(i) Procese și construirea proceselor

Programele OCCAM pot fi descrise ca procese care execută acțiuni și apoi se termină. Conceptul de proces poate fi interpretat la mai multe nivele în program. Într-adevăr, programul întreg poate fi considerat ca un proces care începe, execută anumite acțiuni și apoi se încheie. La nivelul cel mai de jos și primitivele limbajului sînt considerate ca procese, de unde denumirea de procese primitive. Procesele compuse pot fi construite cu primitive sau alte procese printr-un număr de construcții de procese. Scopul lor este indicat în textul programului de o schemă fixă, marcată de două spații. De exemplu,

```
SEQ
  A
  B
```

se citește : execută în secvență întîi procesul A și apoi procesul B. Un constructor similar este folosit pentru exprimarea execuției paralele a proceselor :

```
PAR
  A
  B
```

Se citește : execută procesele A și B în paralel. Ambele procese compuse se încheie numai după ce toate procesele lor constituente s-au încheiat. Totuși, dacă procesul A nu s-a încheiat în cadrul schemei SEQ, procesul B nu va fi lansat în execuție ; în timp ce la construcția PAR, dacă procesul A nu s-a încheiat, procesul B va avea încă o șansă să se execute și termine.

Constructorii paraleli și secvențiali pot fi combinați fără restricții. De exemplu,

```
PAR
  SEQ
    A
    B
  SEQ
    C
    D
```

Aici două procese se execută în paralel; unul este secvența procesului A urmat de B, celălalt este secvența procesului C urmat de D.

Astfel se pot combina procesele, inclusiv primitivele limbajului, pentru a se executa în secvență sau în paralel, acest ultim aspect fiind primul aspect fundamental prezentat în introducere. Ce se poate spune despre regulile privind comunicația între aceste procese? Procesele secvențiale pot partaja memoria, deoarece ele se vor executa pe un procesor și vor reflecta modul de control specific și altor limbaje secvențiale. Procesele componente ale construcției PAR nu pot partaja memoria, chiar dacă se vor executa pe același procesor. În OCCAM, toate comunicațiile dintre procese paralele trebuie să se execute prin intermediul legăturilor de comunicație care conectează procesele respective. Acestea sînt canalele OCCAM. Sînt etichetate și realizează operații de comunicații punct-la-punct fără bufere.

Toate procesele OCCAM active sînt construite pe baza a trei procese primitive, de intrare, ieșire și atribuire. Intrarea și ieșirea sînt primitivele de comunicație folosite în perechi între procese paralele ce folosesc un anumit canal. De exemplu,

```
PAR
  chan ! a + b
  chan ?c
```

Cele două procese primitive paralele comunică între ele prin scrierea (!) și citirea (?) prin canalul 'chan'. Acest proces construit este formal echivalent cu următorul proces de atribuire:

```
c : = a + b
```

Unicul motiv de a scrie această acțiune în paralel ar fi atribuirea de la un proces la altul. Dacă ne gîndim, această transformare de la atribuire la comunicație este una care introduce paralelism plecînd de la o construcție secvențială.

Mai există două procese primitive, dar acestea nu realizează activități utile. Procesul SKIP este lansat, nu realizează nimic și se încheie, fiind folosit cînd sintaxa OCCAM solicită un proces într-o construcție, fără nici o acțiune. Al doilea este procesul STOP. Acesta este lansat, nu face nimic, dar nu se încheie. De exemplu, procesul

```
SEQ
A
B
STOP
C
```

SEQ
A
B
STOP

deoarece procesul C nu se va executa niciodată.

(ii) Sincronizare şi configurare

Ca în CSP, canalul de comunicaţie dintre procesele paralele nu este buferizat. Astfel, oricare proces devine pregătit primul pentru comunicaţie pe un canal, îl va aştepta pe celălalt pînă la executarea primitivei de comunicaţie. Există un handshake implicit în implementarea acestei secvenţe de comunicaţie, deoarece primul proces gata va trebui să semnalizeze celui de-al doilea că este în aşteptare. Pentru detalii de implementare consultaţi §3.5.4. Prin urmare, comunicaţia furnizează o referinţă globală de timp între cele două procese paralele. Dacă sincronizarea nu este importantă pentru execuţia programului, de exemplu într-o situaţie producător-consumator, utilizatorul poate adăuga procese de buferizare. În mod ideal, procesul ar trebui să fie în permanenţă gata să accepte o operaţie de intrare sau ieşire. Desigur, trebuie să se execute în paralel cu procesele producător şi consumator. În continuare este prezentată structura de comunicaţie a unui astfel de program. Dacă buferul are spaţu suficient corespunzător traficului dintre producător şi consumator, procesele se pot executa fără a cunoaşte unul de existenţa celuilalt.

```
PAR
— producător
out !...
— buffer
PAR
out ?...
in !...
— consumator
in ?...
```

Se spune că un program OCCAM este configurat cînd canalele OCCAM sînt atribuite legăturilor de comunicaţie fizice ale transputerului. Cînd se realizează aceasta, programul OCCAM asigură o structură statică a procesului, care la un anumit nivel al ierarhiei este distribuit reţelei de transputere. Astfel, programele OCCAM asigură cod pentru execuţia pe transputere, ca şi informaţii privind reţeaua de interconectare. Într-adevăr, adesea este util să se considere procesele OCCAM ca baloane iar canalele OCCAM ca arce ce le leagă. O astfel de reprezentare apare în fig. 4.8.

Protocolul handshake care asigură sincronizarea între transputere este generat de legătura hardware în cursul procesului de comunicaţie. Dacă mai mult de un proces paralel se execută pe un procesor şi un proces aşteaptă un eveniment de sincronizare, atunci acel proces este înlăturat, iar alte procese pot concura la resursele procesorului. La recepţionarea acelui eveniment, procesul suspendat este replanificat. În acest mod se poate

evita încărcarea nedeterministă a transputerelor în sistem, deşi încărcarea statică de ansamblu trebuie decisă de programator. Implementări ulterioare ale OCCAM-ului vor putea relaxa această descriere statică a proceselor.

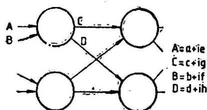


Fig. 4.8 Transformata Fourier rapidă exprimată algoritmic ca o mulţime de procese care comunică. fiecare executînd operaţii de înmulţire şi adunare/scădere complexe asupra datelor de intrare, Reţeaua de comunicaţie va fi eţeaua fluture

```
PROC TWIDDLE(CHAN m,n,o,p VAR w,z)
  VAR ptr
  ptr = 4
  WHILE TRUE
    SEQ
      VAR a[6].b[6].c[6].d[6].t[2]
      PAR
        m?a[ptr for 2]
        n?b[ptr for 2]
        SEQ
          PAR
            t[0] = b[ptr - 2]*w - b[ptr - 1]*z
            t[1] = b[ptr - 2]*z + b[ptr - 1]*w
          PAR
            c[ptr - 2] = a[ptr - 2] + t[0]
            c[ptr - 1] = a[ptr - 1] + t[1]
            d[ptr - 2] = d[ptr - 2] - t[0]
            d[ptr - 1] = d[ptr - 1] - t[1]
        o!c[ptr - 4 for 2]
        o!d[ptr - 4 for 2]
      ptr = (ptr + 2) 6
```

În cazul comunicaţiei între două procese ce se execută pe un singur transputer, canalele OCCAM sînt implementate prin deplasarea datelor în memoria transputerului, cu folosirea aceluiaşi protocol de sincronizare. Mai mult, în ambele situaţii se folosesc aceleaşi ordine, ce permit unui cod precompilat să aibă un canal atribuit unei legături sau locaţii de memorie. Astfel, este posibil să se execute un program OCCAM pe diferite reţele de transputere, prin modificarea cuantelor de timp, dar numai dacă structura procesului este izomorfă cu reţeaua fizică de transputere. Dacă se realizează acest lucru, singurele modificări necesare la codul OCCAM ar fi instrucţiunile care alocă canalele la legături.

(*iii*) *Alegerea şi nedeterminismul*

Ca şi alte limbaje secvenţiale, OCCAM-ul are alţi constructori, care permit o alegere a acţiunii. Primul este cel de buclare. De exemplu,

```
WHILE condiţie
SEQ
A
B
```

Această construcţie se execută în mod repetat pînă ce condiţia de control devine FALSE, moment în care se încheie. În acest exemplu corpul este un proces format din secvenţa de procese A şi B.

Al doilea mecanism convențional pentru alegere în OCCAM este un constructor IF generalizat, care permite definirea proceselor condiționale. Semantica instrucțiunii IF este similară celei de la CASE în limbajul PASCAL. Intrucțiunea IF poate lua orice număr de procese, fiecare trebuind să aibă o condiție de test asociată. Instrucțiunea IF este lansată, evaluează fiecare test în ordinea scrisă și execută primul proces al cărui test a fost evaluat cu valoarea TRUE ; apoi se încheie. Prin urmare, o instrucțiune IF fără nici o componentă se comportă ca un proces STOP. Nu este necesar ca fiecare test să împartă spațiul testat în mulțimi disjuncte, dar pentru a asigura terminarea instrucțiunii IF, spațiul trebuie acoperit. Din acest motiv, adesea o instrucțiune IF se va încheia cu o combinație de TRUE și SKIP. De exemplu,

```
IF
a = 1
A
a = 2
B
TRUE
SKIP
```

Acest cod asigură alegeri pentru a = 1 și a = 2 și acoperă spațiul de test cu condiția TRUE. Datorită semanticii instrucțiunii, procesul SKIP se va executa numai dacă a nu este egal cu 1 și nu este egal cu 2.

Instrucțiunea IF permite o alegere deterministă într-un program OCCAM ; alegerea este determinată de valoarea variabilelor în cadrul domeniului constructorului și semantica execuției ei. Nu se poate spune același lucru dacă alegerea se face pe baza stării canalului, deoarece starea lor nu poate fi determinată la un moment dat, avînd o funcționare asincronă. Astfel, un alt constructor, ALT, realizează alegerea nedeterministă. Constructorul ALT asigură un număr de procese alternative și, precum constructorul IF, fiecare proces are o opțiune care determină execuția lui. Spre deosebire de constructorul IF, alegerile trebuie să includă o intrare pe canal ; de asemenea, trebuie să conțină o condiție (ca la constructorul IF). Opțiunile constructorului ALT sînt denumite sentinele (guards), conform definiției lui Dijkstra (1975).

```
running : = TRUE
          VAR x
          WHILE running
          ALT
          chan 1 ? x and RUNNING
          chan 3 ! x
          chan 2 ? x and RUNNING
          chan 3 ! x
          onoff ?ANY
          running : = NOT running
```

Programul va multiplexa intrările recepționate de la canalele 1 și 2 pe canalul 3, pînă ce primește informații pe intrarea canalului 'onoff'. Procesul ALT va fi lansat, va aștepta pentru satisfacerea uneia din opțiunile nedeterministe (în acest caz se așteaptă o intrare și evaluarea condiției), execută procesele sentinele și apoi se încheie.

Acest program mai ilustrează o serie de alte aspecte privind limbajul OCCAM; de exemplu, intrarea la ANY asigură un semnal, deoarece datele asociate cu operaţia de comunicaţie sînt eliminate. Numai sincronizarea este relevantă. Să observăm şi modul de terminare a programului. Variabila 'running' trebuie definită ca un proces sentinelă, altfel instrucţiunea ALT s-ar putea să nu se încheie (din acest punct de vedere un proces ALT se comportă ca un proces IF). De exemplu, poate fi resetat după testul WHILE, dar înaintea testului ALT, în care caz nici o sentinelă nu va fi satisfăcută la trecerea procesului WHILE. Programul ilustrează declaraţia de variabilă care poate fi asociată cu orice proces sau construcţie a limbajului, iar domeniul său este determinat de persistenţa procesului. Astfel, variabila x va fi alocată din stiva procesului înaintea execuţiei constructorului WHILE şi folosită după terminare. Într-adevăr este posibil să se realizeze :

```
VAR x
x : = VALUE
```

(*iv*) *Multiplicatori*

Adesea este de dorit folosirea multiplicatorilor în asociere cu constructorii paraleli sau secvenţiali. OCCAM permite acest lucru ca o extensie a sintaxei constructorilor respectivi. De exemplu,

```
SEQ i = 0 FOR 5
A
B
```

va executa secvenţa formată din procesele A şi apoi B de 5 ori. Procesele pot conţine masive care folosesc pe i ca index, sau îl pot folosi ca etichetă în alte moduri, ca în alte construcţii secvenţiale de bucle. Valorile luate de i sînt [0 1 2 3 4]. Folosirea multiplicatorului paralel este mai interesantă deoarece poate fi folosită la descrierea masivelor de procese paralele. De exemplu,

```
PAR i = 0 FOR 52
    PAR j = 0 FOR 32
        A
```

va defini un masiv bi-dimensional de procese A. Variabilele multiplicator pot fi folosite pentru selecţia din masive de canale şi pentru a descrie comunicaţia dintre copiile acestui proces. Trebuie să menţionăm că solicitarea de reţele statice în implementările curente ale OCCAM-ului impune o expresie constantă sau o constantă pentru numărul de multiplicări.

Este evident că în acest text putem realiza doar o introducere în limbajul OCCAM. Recomandăm lucrările INMOS (1984) şi Jones (1985) pentru o introducere în OCCAM I, INMOS (1986) şi Pountain (1986)a pentru o prezentare a OCCAM II şi Hoare (1986) şi Roscoe şi Hoare (1986) pentru o tratare mai teoretică a CSP şi OCCAM.

4.5 Tehnici pentru exploatarea paralelismului

Pentru a încheia acest capitol despre limbaje paralele, vom trece în revistă un număr de tehnici care pot fi aplicate la rezolvarea unei probleme

Alegerea tehnicii va depinde de mai mulţi factori, inclusiv limbajul disponibil, modelul de paralelism şi parametrii săi de implementare. Acelaşi algoritm poate fi implementat în multe cazuri folosind cele trei tehnici descrise aici.

De exemplu, folosind reţele de transputere, modelul este unul de paralelism al procesului, iar parametrul cel mai important care trebuie luat în considerare este lărgimea de bandă a comunicaţiei, discutată în § 3.5.5 Aceasta va determina granularitatea cu care un program (sau datele sale). poate fi partiţionat pentru a fi distribuit pe reţeaua de transputere (presupunînd desigur că programul solicită comunicaţii). Pentru un algortim dat, granularitatea şi deci eficienţa pot fi diferite funcţie de tehnica folosită.

4.5.1 Ferma de procesoare

Aceasta este una din metodele cele mai simple de exploatare a paralelismului. Poate fi folosită în aplicaţii unde un mănunchi de procese pot fi „distribuite" unui procesor disponibil. Aceste procese pot fi identice, dar trebuie să fie independente. De exemplu, în fizica energiilor înalte, există o mare cantitate de informaţii experimentale. Acestea trebuie analizate pentru găsirea evenimentelor semnificative. Odată identificat, fiecare eveniment poate necesita o cantitate mare de prelucrări pentru a se determina dacă este semnificativ sau nu. Aceste operaţii pot fi distribuite unui procesor, în timp ce alte evenimente sînt recunoscute şi prelucrate de alte procesoare. Procesoarele pot fi alocate dintr-o coadă de procesoare disponibile. Uneori această tehnică se mai numeşte paralelism al evenimentelor.

Există şi alte aplicaţii unde poate fi exploatată această tehnică, de exemplu prelucrarea formelor independente în cazul proiectării măştilor de circuite integrate, sau calculul pixelilor color independenţi în cazul calculului Mandlebrot. În toate cazurile, alocarea unităţilor de lucru de o mărime suficientă la un procesor va fi realizată de un proces supervizor, a cărui unică sarcină ar fi distribuirea calculelor şi colectarea rezultatelor.

Eficienţa acestei metodologii impune existenţa unor unităţi independente de prelucrare care nu au nevoie de comunicaţii (semnificative), în termenii operaţiilor pe care le execută; în comparaţie cu overhead-ul pentru distribuirea calculelor şi colectarea rezultatelor. Operaţiile de comunicaţie care sînt totuşi necesare ar trebui să se suprapună cu cele de calcul. Încărcarea supervizorului va limita granularitatea unităţii de calcul.

În fig. 4.9(a) se prezintă structura proceselor ce pot fi implementate pe ferme de procesoare. Fig. 4.9(b) prezintă un model cu un singur nivel, unde activitatea este distribuită pe o structură liniară, sau chiar continuă, rezultatele fiind conectate pe canale. Fiecare proces, fizic distribuit, va conţine cod pentru recepţia şi transmisia de activităţi (ce vor depinde de schemele de bufere folosite) şi o copie a programului care se execută.

Fig. 4.9(c) prezintă o structură ierarhică pentru o fermă de procesoare unde toate nodurile, mai puţin frunzele, vor conţine procese supervizor pentru recepţia şi distribuirea activităţilor, ca şi o copie a programului executat. Nodurile frunză au nevoie numai de un proces de buferizare plus programul.

4.5.2 Paralelismul algoritmic

A doua tehnică este legată de structurile pipeline. În acest caz, un program sau proces este partiționat și distribuit rețelei de procesoare sau agenților de calcul. Fiecare partiție se execută în paralel cu datele sale sau cu cele primite de alte părți ale programului. În cazul existenței unor limite secvențiale la evaluarea partițiilor, se poate folosi conceptul de pipelining pentru menținerea concurenței. Cînd este folosit, solicită un flux continuu de date prelucrat de fiecare partiție, astfel că orice inițializare impusă de secvență să fie amortizată.

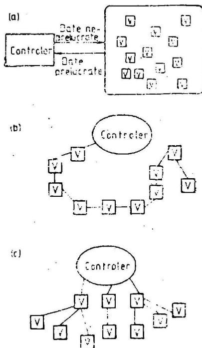


Fig. 4.9 Conceptul fermei de procesoare. (a) Modelul fermei de procesoare.. (b) O rețea liniară de procese ale fermei. (c) O rețea arbore de procese ale fermei.

Rețeaua de procesoare necesară în acest caz trebuie să reflecte fluxul de date pentru algoritmul dat. De fapt, trebuie să formeze un graf static al algoritmului. Granularitatea proceselor distribuite poate fi-teoretic — la nivelul unei operații, ca la mașinile data flow dinamice (Gurd și Watson 1980, Watson și Gurd 1982), sau chiar mai fină dacă considerăm microprogramarea arhitecturilor pipeline. Granularitatea poate fi la nivelul programului, ca în cazul proceselor pipeline UNIX. Granularitatea proceselor trebuie determinată de raportul timpului cerut pentru calcule în comparație cu cel necesar execuției operațiilor de comunicație a datelor la un proces dat.

Acesta este un mod foarte eficient de programare a transputerului, în special dacă codul și datele necesare în fiecare transputer sînt mici în comparație cu capacitatea memoriei sale interne, deoarece aceasta înseamnă un sistem cu o mare densitate a puterii de calcul și o densitate mică a memoriei sale interne, deoarece aceasta înseamnă un sistem cu o mare densitate a puterii de calcul și o densitate mică a memoriei distribuite. Într-adevăr, dacă partițiile programului sînt suficient de mici, programul întreg și datele sale se pot păstra în cei 4 KB ai unui T800.

Această tehnică a fost exploatată adesea în hardware-ul specializat, de exemplu la prelucrarea semnalelor numerice. Cu o componentă programabilă, cum este transputerul, multe aplicații pot exploata același hardware. Problema este că, dacă rețeaua nu reflectă cu acuratețe graful de evoluție al procesului de calcul, granularitatea proceselor distribuite va deveni inacceptabil de mare. Motivul este întîrzierea suplimentară, overhead-ul mai mare și lărgimea de bandă mică asociată cu operațiile de

comunicație între transputerele care nu sînt vecine. Un exemplu de implementare FFT este prezentat în fig. 4.8, unde apare o submulțime a rețelei procesului, împreună cu fragmentul din programul OCCAM 1 care se execută pe fiecare transputer.

4.5.3 Paralelismul geometric

Ultima tehnică de exploatare a multiplicării implică o partiționare a datelor și nu a programului. Dacă un algortim dat poate prelucra concurent o structură mare de date, atunci poate fi partiționată pe o rețea de procesoare. În acest caz, algoritmul folosit pentru prelucrarea datelor va fi copiat în fiecare procesor al rețelei. Apoi, algoritmul se va aplica partiției locale din structura de date, împreună cu orice alte date comunicate de alte partiții.

Este important, datorită motivelor expuse în §3.3, ca rețeaua să reflecte schemele de comunicație necesare între partițiile structurii de date.

5

ALGORITMI PARALELI

5.1 Principii generale

Pentru a atinge performanța optimă a fiecărui calculator este necesar ca în scrierea programului să se țină cont de arhitectura calculatorului. Acest imperativ valabil pentru calculatoarele seriale se păstrează și la cele paralele. Acesta este motivul pentru care programele scrise în limbaj de asamblare, care țin cont de structura calculatorului, pot încă depăși performanțele codului produs de cele mai sofisticate compilatoare. Cu apariția calculatoarelor paralele s-a schimbat doar raportul dintre performanța unui program bun și cea a unui program slab. Acest raport nu depășește probabil un factor de 2 sau 3 pentru un calculator serial, în timp ce factori de 10 sau mai mult nu sînt neobișnuiți la un calculator paralel. În mod destul de simplu, importanța stilului de programare a crescut substanțial.

În acest capitol nu facem nici o încercare de a prezenta algoritmii paraleli pentru toate domeniile importante ale analizei numerice — o astfel de sarcină ar impune un volum separat. În loc de aceasta am selectat o serie de probleme înrudite care ilustrează tipul de considerații ce sînt probabil importante în alegerea unui algoritm și arătăm cum poate fi analizată performanța relativă a algoritmilor într-un mod simplu, folosind conceptele prezentate în cap. 1. Începem în § 5.2 cu problema simplă a calculării sumei unei mulțimi de numere și extindem apoi tehnicile la rezolvarea recurenței generale de ordinul 1. Înmulțirea matricelor (§5.3) este o altă problemă simplă care evidențiază o varietate de soluții, fiecare din ele adecvată unui tip diferit de calculator paralel. Sistemele de ecuații tridiagonale (§5.4) și transformatele (§5.5) intervin foarte frecvent și merită o tratare specială. Rezultatele obținute în aceste ultime două secțiuni sînt folosite ulterior în §5.6 pentru a analiza diferite metode iterative sau directe pentru rezolvarea ecuațiilor cu derivate parțiale. Această secțiune pune un accent deosebit pe metodele de transformare rapidă care se aplică unor anumite clase de ecuații diferențiale parțiale simple, ca de exemplu ecuația Poisson. Dintre multele domenii pentru care nu avem spațiu de tratare, cele mai importante sînt probabil optimizarea, găsirea rădăcinii, ecuații diferențiale ordinare, ecuații liniare generale complete sau rare, inversarea matricelor și determinarea valorilor proprii. Pentru unele din

aceste subiecte recomandăm lucrările: Miranker (1971), Poole și Voight (1974), Heller (1978), Sameh (1977), ca și referințele citate de acești autori. Alte perspective asupra calculului paralel prezintă Kulisch și Miranker (1983) și Kung (1980).

Analiza unui algoritm paralel trebuie efectuată în cadrul unui anumit model de evaluare a timpului de calcul. Noi folosim aici modelul (r_∞, $n_{1/2}$) comportării hard-ului în timp (§1.3.2) pentru a dezvolta metoda $n_{1/2}$ de analiză a algoritmului pentru calculul SIMD (§ 5.1.6) și metoda de analiză $s_{1/2}$ pentru calculul MIMD (§ 5.1.7). Mai există multe alte modele, de exemplu cel al lui Bossavit (1984) pentru calculul vectorial și al lui Kuck (1978) și Calahan (1984) pentru calculul MIMD.

Pînă în anul 1984 nu a existat nici o rezervă dedicată în exclusivitate publicării algoritmilor pentru calculatoarele paralele. Cele mai multe articole au apărut în S.U.A. în *IEEE Transaction on Computers*, în *Journal* și *Communications of the ACM*, *sau în Journal of Computational Physics* (Academic Press). În Europa principala sursă este *Computer Physics Communications* (North Holland), care a publicat lucrările conferințelor *Vector and Parallel Processors in Computational Science* (VAPP 1982, 1985) ce au avut loc la Chester în 1981 și la Oxford în 1984. Alte lucrări ale unor conferințe europene ce conțin algoritmi paraleli sînt *Conpar'81* (Handler 1981) și *Parallel Computing '83, '85* (Feilmeier, Joubert și Schendel 1984, 1986). Conferința principală din S.U.A. este *International Conference on Parrallel Processing*, care se ține anual și ale cărei lucrări sînt publicate de IEEE Computer Society Press. În 1984 au apărut două reviste noi dedicate în întregime problemelor calculului paralel; acestea sînt *Parrallel Computing* (North Holland) și *Journal of Parallel and Distributed Computing* (Academic Press). Revista *Computer and Artificial Intelligence*, publicată începînd cu 1982 de Academia slovacă de științe, cu lucrări în engleză și rusă, acperă și domeniul algoritmilor paraleli. Alte surse principale pentru algoritmi paraleli sînt colecțiile de lucrări editate de cunoscuți specialiști în domeniu: *High Speed Computers and Algorithm Organization* (Kuck et al 1977); *Parallel Computations* (Rodrigue 1982); *Parallel Processing System* (Evans 1982); *High Speed Computation* (Kowalik 1984); *Algorithms, Software and Hardware of Parallel Computers* (Miklosko and Kotov 1984); *Distributed Computing* (Chambers et al 1984); *Distributed Computing Systems Programme* (Duce 1984); *Introduction to Numerical Methods for Parallel Computers* (Schendel 1984); *Parallel MIMD Computation* (Kowalik 1985); *Vector and Parallel Processors for Scientific Computation* (Squazzero 1986); și *Scientific Computing on Vector Computers* (Schonauer 1987). Se pare, după creșterea explozivă a publicațiilor începînd cu 1984, că domeniul de studiu al calculului paralel a devenit de sine stătător.

5.1.1. Performanța algoritmilor

Vom defini performanța unui program de calculator ca fiind invers proporțională cu timpul CPU (timp unitate centrală, timpul cît unitățile de calcul — (aritmetică și logică) — sînt folosite de un program; nu includ timpul pentru operațiile de I/E — consumat pentru execuția programului).

5.1.2 Paralelismul

În orice etapă a execuției unui algoritm, paralelismul său este definit ca numărul operațiilor aritmetice ce sînt independente și pot fi executate prin urmare în paralel, cu alte cuvinte concurent sau simultan. La un calculator pipeline operanzii vor fi definiți ca vectori, iar operația va fi executată cu o singură instrucțiune vectorială. Paralelismul este același cu lungimea vectorului. La un masiv de procesoare datele pentru fiecare operație sînt alocate diferitelor elemente procesoare ale masivului, iar operațiile sînt executate în același timp ca rezultat al interpretării unei instrucțiuni în unitatea centrală master. În acest caz, paralelismul este egal cu numărul operanzilor ce sînt prelucrați în paralel, în acest mod. Paralelismul poate rămîne în cursul diverselor etape ale unui algoritm (ca în cazul înmulțirii matricelor, §5.3) sau poate varia de la etapă la etapă (ca în cazul *SERICR*, forma serială a reducerii ciclice, §5.4.3).

Adesea este aleasă arhitectura calculatoarelor paralele pentru atingerea performanței celei mai bune cînd se execută operații cu vectori de anumite lungimi (adică, cu un anumit număr de elemente). Acesta este, prin definiție, paralelismul hardware natural al calculatorului. De exemplu, ICL DAP 64×64 pune la dispoziție trei tipuri de memorare și moduri de operații aritmetice cu vectori, respectiv, de lungime 1 (memorare orizontală și mod scalar), de lungime 64 (memorare orizontală și modul vectorial și lungime 4096 (memorare verticală și mod matrice). Deși aceste moduri sînt disponibile prin program, ele sînt alese pentru conformare la dimensiunea hardware a masivului DAP și constituie astfel trei nivele de paralelism, fiecare cu nivelul său propriu de performanță. La calculatoarele pipeline fără registre vectoriale, ca de exemplu *CYBER 205*, performanța medie (ecuațiile 1.9 și figura 1.16) crește monoton odată cu creșterea lungimii vectorului, iar cineva poate să spună că paralelismul hardware natural este cît se poate de mare (limita superioară este reprezentată de lungimea maximă a vectorului permis de hard-ul mașinii, adică 64K—1). La calculatoarele pipeline cu registre vectoriale, ca de exemplu CRAY X—MP, cea mai bună performanță se atinge pentru vectori cu lungimea multiplu de numărul elementelor înregistrate într-un registru vectorial. În cazul calculatorului CRAY X—MP cu registre vectoriale de 64 elemente, paralelismul natural este 64 sau multipli ai acestui număr.

Obiectivul unui bun programator/analist în calculul numeric este găsirea unei metode de rezolvare care să realizeze cea mai bună concordanță între paralelismul algoritmului și paralelismului natural al calculatorului.

5.1.3 Paracalculatorul și eficiența

Pentru a stabili performanța masivelor de procesoare, Schwartz (1980) a propus conceptul de paracalculator. Acest calculator este un masiv SIMD infinit de elemente procesoare, fiecare din ele avînd acces la o memorie comună în paralel pentru fiecare dată. Orice algoritm ar realiza performanța maximă pe un astfel de calculator, deoarece se elimină cauzele obișnuite de ineficiență. Paracalculatorul nu are întîrzieri de transfer sau conflicte de acces la memorie (vezi §5.1.4 și §5.1.5) și are întotdeauna suficiente ele-

Cu alte cuvinte, un program de mare performanţă îşi atinge obiectivul în timpul cel mai scurt. În acest context, obiectivul urmărit este minimizarea timpului CPU folosit pentru obţinerea soluţiei problemei. Aceasta nu este singura definiţie a performanţei care ar putea fi dată — s-ar putea impune costul minim pentru instalarea unui anumit calculator sau minimizarea spaţiului de memorie folosit. Oricum, creşterea performanţei în cadrul definiţiei noastre este de obicei dezirabilă şi este cel mai uşor de cuantificat. Orice referinţă la cost este subiectul inconstanţei algoritmilor particulari de încărcare, iar disponibilitatea unei memorii mari foarte rapide (256 Mw pare să fi devenit norma la calculatoarele paralele) înseamnă că utilizarea memoriei nu mai este probabil o restricţie importantă.

Este interesant de notat că la prima generaţie de calculatoare, de exemplu EDSAC care avea un total de 512 cuvinte de memorie rapidă (Wilkes et al 1951), minimizarea utilizării memoriei a fost criteriul de proiectare cel mai important. În consecinţă, au fost elaboraţi algoritmi speciali pentru a-l respecta, de exemplu varianta Gill pentru metoda Runge-Kutta de rezolvare a ecuaţiilor diferenţiale ordinare (Gill 1951). De asemenea, este important de observat că, la calculatoarele paralele, obiectivul timpului de execuţie minim nu este în mod necesar sinonim cu realizarea numărului minim de operaţii aritmetice ca în cazul calculatoarelor seriale. Exemplele prezentate în acest capitol vor evidenţia acest lucru, ce apare deoarece cîştigurile în viteză produse de creşterea numărului de operaţii executate în paralel poate cîntări mai mult decît costul introducerii unor operaţii aritmetice suplimentare. Astfel, deşi măsurarea performanţei hardware a unui calculator prin numărul de operaţii în virgulă mobilă executate pe secundă este sensibilă, o astfel de măsură singură nu este în mod necesar folositoare în alegerea unui algoritm pentru un calculator paralel (vezi §5.1.6 şi § 5.1.7).

Performanţa unui program de calculator depinde atît de adecvarea procedurii numerice — cunoscută ca algoritm — folosită pentru rezolvarea problemei cît şi de îndemînarea cu care este implementat algoritmul pe calculator de programator sau compilator în timpul operaţiei de codare. În acest capitol examinăm algoritmii adecvaţi pentru rezolvarea unei game de probleme obişnuite folosind calculatoarele paralele. Dacă paralelismul algoritmului corespunde paralelismului calculatorului, este aproape sigur că un program de performanţă ridicată poate fi scris de un programator experimentat. Totuşi, considerarea detaliilor de programare pentru diverse calculatoare este în afara scopului acestei cărţi, iar cititorul este îndrumat către manualele de programare ale calculatorului său particular. Cele mai multe calculatoare paralele dispun de un compilator vectorizant pentru un limbaj de nivel înalt, de obicei *FORTRAN* cu sau fără extensii de prelucrare a masivelor (vezi cap. 4) şi trebuie să fie posibilă atingerea a cît mai mult din performanţa potenţială într-un astfel de limbaj. În cele mai multe cazuri va fi necesară programarea părţilor cheie ale programului în limbaj de asamblare — dispunînd astfel de toate posibilităţile calculatorului — dacă trebuie obţinută performanţa maximă (de exemplu performanţa supervectorială la CRAY X-MP vezi cap. 2).

mente procesoare. Deşi paracalculatorul este irealizabil practic, este un concept folositor pentru stabilirea performanţei algoritmilor şi a masivelor de procesoare actuale. De exemplu, se poate defini paraeficienţa ca:

$$\varepsilon_p = \frac{\text{timpul de execuţie pe paracalculator}}{\text{timpul de execuţie pe un calculator actual}} \tag{5.1}$$

unde presupunem că elementele procesoare au aceeaşi performanţă hardware pe paracalculator şi calculatoarele actuale.

De asemenea, se poate folosi raportul dintre timpul de execuţie a 2 algoritmi pe paracalculator, ca o măsură relativă a performanţei lor. Totuşi, această măsură poate să nu fie un indicator bun al performanţei relative pe calculatoarele actuale, deoarece ignoră timpul necesar transferului datelor între elementele procesoare aflate la distanţă în masiv (întîrzieri de transfer). Grosch (1975) a utilizat conceptul paracalculatorului pentru compararea performanţelor diferiţilor algoritmi pentru rezolvarea ecuaţiei Poisson pe masive de procesoare cu diferite structuri de interconectare între elementele procesoare. El compară topologia vecinătăţii maxime, utilizată pe ICL DAP (vezi §3.4.2) cu şi fără transferul la distanţă mare realizat de topologia amestecului perfect aşa cum a fost propusă de Stone (1971).

Deşi inventat pentru compararea masivelor de procesoare, conceptul paracalculatorului poate fi utilizat în studiul procesoarelor pipeline. Paracalculatorul corespunde unui procesor pipeline cu timpul de iniţializare 0, fără conflicte de acces la memorie şi $n_{1/2}$ de valoare infinită. Reamintindu-ne că $n_{1/2} = s + l - 1$ (vezi ec. 1.6.b) unde $s\tau$ este timpul de iniţializare şi l numărul subfuncţiilor care se suprapun, se poate vedea că, dacă paracalculatorul tinde spre l, proporţia funcţionării paralele a pipelineului devine mare. La cealaltă extremă, se poate defini un procesor serial perfect ca un procesor pipeline cu numai o subfuncţie (de exemplu, unităţile aritmetice nu sînt segmentate) şi care, de asemenea, nu are timp de iniţializare sau conflicte de acces la memorie. Procesorul serial perfect corespunde, prin urmare, unui procesor pipeline cu $n_{1/2} = 0$. Actualele procesoare pipeline, cu un număr finit de subfuncţii şi, prin urmare, o valoare finită a lui $n_{1/2}$, se află între procesoarele seriale perfecte şi paracalculatoar, funcţie de lungimea performanţei jumătate. Astfel caracterizat, obţinem spectrul calculatoarelor descris în §1.3.4 şi prezentat în fig. 1. 15.

5.1.4 Întîrzieri datorate operaţiilor de transfer

Am menţionat deja că timpul transferului informaţiilor între diferitele elemente procesoare ale unui masiv poate avea un efect important asupra para-eficienţei algoritmilor implementaţi pe astfel de calculatoare. Aceste întîrzieri pot fi relativ neimportante, dacă timpul pentru execuţia unei operaţii aritmetice este mult mai mare decît timpul pentru transferul informaţiei între elementele procesoare ale unei perechi, cum este cazul pentru

operațiile in virgulă mobilă la ICL DAP. Oricum, dacă timpul de execuție a operației aritmetice este comparabil cu cel pentru transfer ultimul joacă un rol important în determinarea performanței unui algoritm și nu poate fi ignorat. Această situație intervine la ICL DAP cînd se folosesc cuvinte scurte și operații aritmetice cu întregi, cum se întîmplă probabil la prelucrarea imaginilor. De exemplu, Jesshope (1980a) a arătat că, într-o implementare a transformatei teoretice a numerelor la ICL DAP, întîrzierile datorate transferurilor pot reprezenta mai mult de jumătate din timpul de execuție a algoritmului. Alte rezultate privind transferul informațiilor sînt prezentate în Jesshope (1980b, c).

5.1.5 Conflictele de acces la blocurile de memorie

Întîrzierile de transfer reprezintă problema aducerii într-un procesor a datelor din diferite părți ale memoriei. O problemă similară pentru un calculator vectorial pipeline tipic, precum CRAY X—MP, este aceea a conflictului de acces la memorie care poate interveni cînd datele sînt aduse din memoria organizată în blocuri a unei astfel de mașini, la una din unitățile aritmetice pipeline. Deși memoria unor astfel de mașini este descrisă adesea ca o memorie RAM de mai multe Mw, această prezentare este înșelătoare. Ar fi aproape imposibil să se asigure o conexiune paralelă separată între fiecare cuvînt de memorie și unitățile pipeline. În practică aceste memorii sînt împărțite într-un număr relativ mic de blocuri independente. La CRAY X—MP cu o memorie de 2 Mw există 16 blocuri a 128 Kw, fiecare bloc avînd posibilitatea să servească simultan o cerere distinctă de acces la memorie. Blocurile sînt numerotate de la 0 la 15 și ordonate ciclic în secvența numerelor cu blocul 0 urmînd blocului 15. Un vector continuu este un vector ale cărui elemente sînt memorate în blocuri succesive și prin urmare diferite. Prin urmare aceste elemente pot fi accesate la cicli succesivi ai ceasului. Aceasta este viteza maximă de livrare a datelor (descrisă ca o lărgime de bandă de 1 cuvînt pe o perioadă de ceas) și coincide cu viteza maximă la care o unitate aritmetică pipeline poate accepta operanzi. Intervin întîrzieri dacă viteza de transfer este mai mică decît cea maximă. Aceasta se întîmplă cînd se efectuează o cerere de acces la memorie la un bloc care este încă ocupat cu servirea unei cereri anterioare. Acest fenomen este cunoscut drept conflictul de acces la un bloc de memorie.

La CRAY X—MP un bloc de memorie este ocupat 4 perioade de ceas cînd servește o cerere de acces, iar o nouă cerere nu poate fi acceptată în acest interval de timp. Totuși, cererile la blocurile diferite care nu sînt ocupate pot fi efectuate în cursul unor perioade de ceas succesive. Este evident că fiecare al patrulea element al unui vector contiguu poate fi accesat la viteza maximă deoarece cererile la oricare bloc intervin la intervale de 4 perioade de ceas, tocmai cînd blocul a finalizat cererea anterioară. Dacă totuși s-au efectuat cereri în cursul unor perioade de ceas succesive la fiecare al optulea element al vectorului, ele vor ajunge la fiecare bloc la fiecare 2 perioade de ceas în timp ce blocul este încă ocupat cu cererea anterioară. Prin urmare, fiecare al optulea element al vectorului poate fi accesat numai la jumătate din viteza maximă, deși valorile succesive sînt înregistrate în

blocuri de memorie diferite. Fiecare al 16-lea element al vectorului este memorat în acelaşi bloc de memorie şi valorile succesive pot fi accesate în cel mai bun caz la 4 perioade de ceas, adică la un sfert din viteza maximă.

Prin urmare, este evident că în selectarea unui algoritm pentru o maşină cu o memorie organizată pe blocuri, conflictele de acces la memorie trebuie evitate. Din păcate mulţi algoritmi numerici buni se bazează pe înjumătăţirea sau dublarea recursivă şi implică referinţe succesive la date aflate în locaţii de memorie separate printr-o putere a lui 2 (de exemplu, transformata Fourier rapidă, vezi § 5.5.2). De asemenea, se obişnuieşte să se înmulţească matrici al căror număr de linii sau coloane este o putere a lui 2. Deoarece, de obicei numărul blocurilor este ales ca o putere a lui 2, conflictele de acces la memorie pot interveni foarte uşor. Acesta este un motiv pentru neatingerea performanţei supervectoriale de către CRAY—1.

Astfel de conflicte pot fi minimizate prin considerarea atentă a modului de memorare a matricei. Să considerăm o matrice 16 × 16 înregistrată coloană cu coloană într-o memorie formată din 16 blocuri. Compilatorul FORTRAN foloseşte această strategie de memorare în mod obişnuit. Elementele succesive ale oricărei coloane sau ale diagonalei principale sau secundare pot fi accesate la viteza maximă deoarece sînt înregistrate în blocuri de memorie diferite. Totuşi, toate elementele aceleiaşi linii sînt înregistrate în acelaşi bloc şi nu pot fi accesate succesiv pentru operaţii cu linii fără conflict de acces. Această problemă poate fi rezolvată prin program prin adăugarea unei linii şi coloane false la matrice. Aceasta va fi apoi memorată ca o matrice 17 × 17. Elementele succesive ale oricărei coloane (interval de memorare de 1 bloc), linie (interval 17 blocuri), sau diagonalei principale (interval 18 blocuri) pot fi accesate fără conflict. Numai accesele la diagonala secundară (interval de memorare 16 blocuri) conduc la conflict de acces, dar acest mod este rar în cazul operaţiilor cu matrici.

Conflictele de acces la memorie pot fi minimizate în modul menţionat anterior, dacă numărul blocurilor de memorie este ales un număr prim. Calculatorul BSP utilizează această soluţie hardware, avînd 17 blocuri de memorie. În exemplul de mai sus al matricei 16 × 16, toate liniile, coloanele şi diagonalele pot fi accesate fără conflict. O caracteristică importantă a unui număr prim de blocuri de memorie (altul decît 2) este că referinţele la elementele unui vector, separate printr-o putere a lui 2, cum intervin repetat în algoritmii cu dublare succesivă, nu pot fi adresate aceluiaşi bloc de memorie.

5.1.6 Metoda $n_{1/2}$ de analiză a algoritmului vectorial (SIMD)

Apariţia noii generaţii de calculatoare vectoriale pune analistului noi probleme. După ce s-a răspuns satisfăcător întrebărilor privind convergenţa numerică şi precizia rezultatului, rămîne problema alegerii celui mai bun algoritm pentru rezolvarea unei anumite probleme pe un calculator particular. Dacă considerăm că „cel mai bun" este similar cu timpul de execuţie cel mai mic, atunci este necesar să ţinem seama de caracteristicile de viteză ale hard-ului şi soft-ului asociat. Considerăm că *metoda $n_{1/2}$ de analiză a algoritmilor* este un mod raţional de a introduce aceste caracteristici în metoda de alegere (Hockney 1982, 1983, 1984b). Parametrul $n_{1/2}$ a

mai fost folosit pentru evaluarea performanţelor de către Arnold (1983), Gannon şi Van Rosendale (1984), Neves (1984), Strakos (1985, 1987) şi Schönauer (1987).

Pentru a evalua timpul de execuţie al unui calculator serial, este necesar să cunoaştem numai numărul total de operaţii aritmetice — denumit adesea ca activitate (work) — deoarece timpul de calculator este direct proporţional cu această cantitate. În acest caz optimizarea este directă deoarece a minimiza timpul este acelaşi lucru cu minimizarea numărului total de operaţii aritmetice, condiţie folosită în mod tradiţional de cînd s-a construit primul calculator von Neuman. La un calculator vectorial, situaţia este totuşi mult mai complexă deoarece o parte din activitatea este reprezentată de un număr mai mic de operaţii vectoriale, unde multe operaţii aritmetice sînt executate în paralel fie de o anumită unitate pipeline, fie simultan de mai multe unităţi aritmetice. Cu fiecare operaţie vectorială se asociază un timp de overhead. Acest overhead este proporţional cu parametrul $n_{1/2}$ al calculatorului şi trebuie inclus în comparaţie. De fapt, calculatoarele vectoriale diferă adesea mai mult prin numărarea acestui overhead decît prin performaţa asimptotică. De aceea, la estimarea timpului de execuţie al unui algoritm pe un calculator vectorial este necesar să se cunoască, şi să se includă, atît efectul numărului de operaţii aritmetice, cît şi al numărului de operaţii vectoriale. Metoda $n_{1/2}$ de analiză a algoritmilor asigură o metodologie pentru includerea ambelor cantităţi în analiza de timp, rezolvînd astfel problema mult mai dificilă a minimizării timpului la un calculator vectorial. Condiţia tradiţională de minimizare a numărului de operaţii aritmetice este, simplu, incorectă şi, în particular, vom vedea că algoritmii cu numărul de operaţii artimetice minim nu se execută în mod necesar într-un interval de timp minim.

(i) Vectorizarea

Prima etapă în modificarea unui program existent, pentru a fi executat pe un calculator paralel, constă în reorganizarea codului, astfel încît cît mai multe bucle DO să fie înlocuite de instrucţiuni vectoriale în timpul procesului de compilare, prin folosirea unui *compilator vectorizant*. Procesul vectorizării poate fi tot ce se poate face, şi lasă programul compus din două părţi : o parte scalară ce va fi executată de unitatea scalară a calculatorului, şi o parte vectorială ce va fi executată de unitatea vectorială. Adesea r_∞ al unităţii scalare este de 10 ori mai mic decît r_∞ al unităţii vectoriale, astfel că timpul de execuţie al algoritmului va depinde în primul rînd de dimensiunea părţii scalare din cod (vezi §1.3.5 şi fig. 1.19). Există mulţi algoritmi asociaţi în mod particular cu rezolvarea ecuaţiilor cu derivate parţiale, în care toate operaţiile aritmetice în virgulă mobilă pot fi executate cu instrucţiuni vectoriale, din care vom prezenta cîteva exemple în acest capitol. Pentru aceşti algoritmi cea mai bună organizare a vectorilor (adică alegerea elementelor ce-i compun şi a lungimii lor) este de o mare importanţă. Se prezintă metoda $n_{1/2}$ de analiză a algoritmilor vectoriali ca o tehnică pentru optimizarea raţională a algoritmilor vectoriali, sau a părţilor vectoriale a codurilor mai complexe care posedă în mod inevitabil şi părţi scalare.

În monografia lui Gentzsch (1984) se trec în revistă tehnicile de vectorizare cu referiri la **CRAY−1, CYBER 205, HITACHI S9** cu **IÀP, ICL DAP** şi Denelcor **HEP**. Alte lucrări cu acest subiect sînt cele ale lui Wang (1980), Swarztrauber (1982), Arnold (1983) Bossavit (1984) şi Neves (1984). Cel mai bun vectorizator automat cunoscut este probabil cel realizat la Universitatea Illinois de Kuck (1981) şi colaboratorii săi, denumit sistemul *parafrase*. Yasamura et al (1984) şi Schonauer (1987) analizează tehnicile de vectorizare, iar performanţele a trei vectorizatori automaţi, inclusiv parafrase, sînt comparate de Arnold (1982).

(*ii*) *Evaluarea timpului de execuţie al algoritmului*

Presupunerea cea mai simplă privind evaluarea timpului de execuţie este că algoritmul vectorial constă dintr-o secvenţă de instrucţiuni vectoriale şi că timpul de execuţie al unei instrucţiuni vectoriale depinde liniar de lungimea vectorului şi poate fi exprimat pe baza parametrilor $(r_\infty, n_{1/2})$ cu formula (1.4a) :

$$t = r_\infty^{-1}(n + n_{1/2}) \tag{5.2}$$

Timpul T consumat pentru execuţia algoritmului este :

$$T = \sum_{i=1}^{i=q} r_\infty^{-1}(n_i + n_{1/2}) \tag{5.3}$$

unde q este numărul operaţiilor vectoriale ce constituie algoritmul, iar n_i este lungimea vectorului în cazul operaţiei vectoriale a i-a. Dacă parametrii r_∞ şi $n_{1/2}$ sînt aproximativ constanţi pentru toate operaţiile, sau se consideră valori medii corespunzătoare, r_∞ şi $n_{1/2}$ pot fi scoşi în factori :

$$T = r_\infty^{-1}(s + n_{1/2} \cdot q) \tag{5.4a}$$

unde

$$s = \sum_{i=1}^{i=q} n_i$$

este numărul total de operaţii aritmetice ale algoritmului.

În alt mod, formula (5.4a) poate fi rescrisă în termenii lungimii medii a vectorului, n :

$$T = r_\infty^{-1} q(\bar{n} + n_{1/2}) \tag{5.4b}$$

sau ca

$$T = r_\infty^{-1} s[1 + (n_{1/2}/\bar{n})] \tag{5.4c}$$

unde $n = s/q$. Expresia din paranteze pătrate a execuţiei (5.4c) este factorul prin care o analiză tradiţională a complexităţii seriale, $r_\infty^{-1}s$, furni-

zează erori cînd se aplică în cazul vectorial. De asemenea, arată că nu valoarea absolută a lui $n_{1/2}$ este importantă, ci raportul ei la lungimea medie a vectorului : $n_{1/2}/\bar{n}$.

(iii) Complexitatea serială şi paralelă

Ecuaţia (5.4a) este interesantă prin aceea că relevă legătura dintre analiza numerică serială tradiţională şi lucrările recente asupra algoritmilor paraleli. Altfel spus, arată legătura între analizele de complexitate serială şi paralelă ale algoritmilor. Un calculator serial are un $n_{1/2}$ mic sau nul, de aici numai *primul* termen al ecuaţiei (5.4a) este important, iar timpul minim de execuţie se obţine prin minimizarea lui s, care indică numărul total de operaţii aritmetice. Analiza algoritmilor paraleli se bazează pe folosirea conceptului de paracalculator, care este caracterizat de $n_{1/2} = \infty$ (vezi § 5.1.3). În acest caz, numai al doilea termen al ecuaţiei (5.4a) este important, iar timpul minim de execuţie se obţine prin minimizarea numărului de operaţii vectoriale, q, *fără* a considera numărul total de operaţii aritmetice implicate. Aceste două perspective contrastante ale optimizării algoritmilor au condus la o dihotomie în comunitatea specialiştilor în proiectarea algoritmilor şi analizei numerice, între cei care „locuiesc" în lumile „serială" şi „infinit paralelă", deoarece pentru rezolvarea aceloraşi probleme se recomandă frecvent algoritmi destul de diferiţi. Considerăm că diferenţele între cele două perspective sînt eliminate prin utilizarea metodei de analiză $n_{1/2}$.

Ambele direcţii ale optimizării algoritmilor sînt extreme nerealiste, deoarece calculatoarele vectoriale actuale au de obicei un $n_{1/2}$ finit, nenul (de exemplu, **CRAY—1** are o valoare aproximativ egală cu 20, iar **CYBER 205** o valoare aproximativ egală cu 100). Intervine atunci întrebarea : „ce trebuie minimizat în cazul unui calculator cu $n_{1/2}$ finit?" Ecuaţia (5.4a) arată că răspunsul constă în aceea că nu trebuie minimizaţi nici s nici q, ci cantitatea ($s + n_{1/2}q$). Metoda de analiză $n_{1/2}$ face tocmai acest lucru. Lungimea performanţei jumătate $n_{1/2}$ este considerată ca parametrul care interpolează liniar între perspectivele extreme serială şi infinit paralelă. În acest mod paralelismul finit al unui sistem de calcul poate fi introdus în analiză. Avantajul principal al metodei este că nimeni nu trebuie să decidă, în mod nerealist, dacă calculatorul este serial sau paralel; gradul de paralelism se introduce odată cu valoarea corectă a lui $n_{1/2}$.

(iv) Diagramele de fază algoritmice

Cînd comparăm algoritmii pentru aceste calculatoare, considerăm raportul celor două expresii de evaluare a timpului de execuţie al ecuaţiei (5.4). Algoritmul este complet descris prin furnizarea valorilor s şi q (sau $\bar{n}$ şi q, sau $\bar{n}$ şi s), iar calculatorul prin cei doi parametri r_∞ şi $n_{1/2}$. În cazul acestui raport, r_∞ se reduce. Astfel, în cadrul aproximărilor acestor analize, $n_{1/2}$ este singura proprietate a calculatorului care influenţează alegerea algoritmilor.

Cînd se compară algoritmi, liniile de egală performanţă (EPL) joacă un rol cheie. Dacă T(a) şi T(b) sînt timpii de execuţie pentru algoritmii (a)

şi (b), atunci performanţa lui (a) este mai mare sau egală cu cea a lui (b), p(a) $\geqslant p$(b), dacă T(a) $\geqslant$ T(b), de unde se obţine

$$n_{1/2} > \frac{s^{(a)} - s^{(b)}}{q^{(b)} - q^{(a)}} \tag{5.5}$$

Inegalitatea (5.5) determină regiunile unui plan unde fiecare algoritm are performanţa cea mai bună, iar egalitatea furnizează formula pentru EPL între algoritmii (a) şi (b).

Datorită naturii caracterizării, $n_{1/2}$ intervine întotdeauna liniar în ecuaţia pentru liniile de performanţă egală. De aceea trebuie trecut explicit în membrul stîng al unei astfel de ecuaţii. Este o proprietate a calculatorului, observată prin compilatoarele şi asambloarele care sînt folosite. Pe de altă parte, membrul drept depinde numai de numărul operaţiilor din cei doi algoritmi, şi poate fi o funcţie neliniară complicată ce depinde de dimensiunea problemei *n.n* poate fi ordinul matricilor într-o problemă cu matrice, sau numărul punctelor într-un caroiaj în cazul unei aproximaţii cu diferenţe finite a unei probleme cu ecuaţii diferenţiale parţiale.

Modul cum se prezintă rezultatele comparaţiei între algoritmi este important. Formule ca (5.5) conţin rezultatul, dar nu pot fi folosite fără un proces de calcul extensiv, iar în cazuri complicate intervin tabele de numere ce nu pot fi folosite. Se preferă, evident, o reprezentare grafică, care ar permite alegerea algoritmului direct pe baza informaţiilor ce definesc problema. O astfel de prezentare se poate face prin trasarea liniilor de performanţă egală între perechi de algoritmi în planul ($n_{1/2}$, n). Aceste linii vor împărţi planul în regiuni în care fiecare algoritm are cea mai bună performanţă. Denumim o astfel de prezentare o „diagramă a fazelor" în analogie cu diagramele folosite în chimia fizică. În cazul diagramei de fază chimică, valorile parametrilor descriu condiţiile (de exemplu, temperatură şi presiune) ce determină un punct în planul care este împărţit în regiuni unde diferitele stări ale materiei au nivelele de energie cele mai mici. Se poate spune că natura alege apoi o stare ca cea mai bună pentru material. În cazul *diagramelor de fază algoritmice*, parametrii ce descriu calculatorul şi dimensiunea problemei determină un punct în planul împărţit în regiuni unde fiecare algoritm are timpul de execuţie cel mai mic. Apoi se alege acest algoritm ca cel mai bun.

Adoptarea anumitor standarde în prezentarea unor astfel de diagrame este avantajoasă, pentru a facilita comparaţiile. De obicei se alege axa x egală sau proporţională cu $n_{1/2}$, iar axa y egală sau proporţională cu dimensiunea problemei n. În acest mod algoritmii convenabili pentru calculatoarele mai seriale ($n_{1/2}$ mic) apar în stînga diagramei, iar cei adecvaţi pentru calculatoarele, mai paralele ($n_{1/2}$ mare) în partea dreaptă. În mod similar, algoritmii buni pentru probleme de mică amploare apar în partea inferioară a diagramei iar cei pentru probleme complexe în partea superioară. Pentru ambele axe este de dorit o axă logaritmică. În fig. 5.3, 5.9, 5.10, 5.26 şi 5.28 (Hockney 1982, 1983) se prezintă cîteva exemple de diagrame de fază algoritmice. În § 5.2.3. intervine un exemplu simplu de construire şi interpretare. Se compară doi algoritmi. În lucrarea lui Hockney (1987a) apare un exemplu mult mai complex pentru 4 algoritmi. Fig. 5.28

prezintă comparaţia a doi algoritmi, furnizînd cea mai bună valoare pentru un parametru de optimizare ce poate fi folosit în planul fazelor.

Am văzut, conform ecuaţiei (5.4c), că raportul $n_{1/2}/\bar{n}$ este mai important chiar decît $n_{1/2}$. Similar, diagramele de fază ale algoritmilor sînt desenate mai compact dacă axa x este egală, cu raportul $n_{1/2}/n$. Acest raport, mai degrabă decît $n_{1/2}$ singur, determină dacă calculele se execută într-un mediu serial (valori mici), sau mediu paralel (valori mari). De asemenea, raportul are avantajul independenţei faţă de unităţile folosite pentru măsurarea lui n.

(*v*) *Algoritmi vectoriali seriali şi paraleli*

În proiectarea algoritmilor vectoriali este avantajoş să se distingă două formulări extreme ale aceleiaşi metode de bază, adică două structuri diferite ale aceloraşi relaţii algebrice/numerice care definesc metoda numerică. Acestea sînt :

(1) *Varianta serială* a algoritmului vectorial, obţinută prin minimizarea numărului total de operaţii aritmetice s — denumirea este dată de faptul că timpul de execuţie va fi minim cînd execuţia se realizează pe un calculator serial, unde $n_{1/2} = 0$;

(2) *Varianta paralelă* a algoritmului vectorial, care se obţine prin minimizarea numărului de operaţii vectoriale q (sau, echivalent, maximizînd lungimea vectorului mediu $\bar{n}$) — această variantă se va executa într-un timp minim pe un calculator infinit paralel, unde $n_{1/2} = \infty$, sau pe un masiv de procesoare care posedă un procesor pentru fiecare element al vectorului cel mai lung.

Analizele ce urmează se limitează la aceste două soluţii. În situaţii mai complexe, pot fi dezirabile variante hibride ale celor două soluţii extreme. De exemplu, dacă într-o problemă lungimea vectorului este mult mai mare decît numărul procesoarelor unui masiv, atunci varianta serială poate fi folosită adesea pentru a reduce lungimile vectorilor la mai puţin decît dimensiunea masivului, după care se poate folosi varianta paralelă.

(*vi*) *Unităţile scalară şi vectorială*

Metoda de analiză $n_{1/2}$, reprezentată ca diagrama de fază logaritmică, poate fi folosită, de asemenea, pentru compararea performanţelor unui algoritm scalar, executat de unitatea scalară a unui calculator, cu performanţa unui algoritm vectorial echivalent, executată de unitatea vectorială a aceluiaşi calculator. Dacă presupunem că algoritmul scalar are $s^{(s)}$ operaţii în virgulă mobilă, atunci timpul de execuţie al algoritmului scalar în unitatea scalară este

$$T_s = s^{(s)}/r_{\infty s} \tag{5.6a}$$

unde $r_{\infty s}$ este performanţa unităţii scalare. Dacă unitatea vectorială este caracterizată de parametrii $((r_{\infty v}, n_{1/2})$, timpul de execuţie al algoritmului vectorial va fi.

$$T_v = (s^{(v)} + n_{1/2}q^{(v)})/r_{\infty v} \tag{5.6b}$$

unde exponentul (v) indice numărul operaţiilor algoritmului vectorial. Formula pentru linia de egală performanţă între cele două alternative este definită de condiţia $T_s = T_v$, deci

$$n_{1/2} = (R_{\infty s}(s) - s^{(v)})/q^{(v)} \qquad (5.7)$$

unde $R_\infty = (r_{\infty v}/r_{\infty s})$ este raportul performanţelor asimptotice ale unităţilor vectorială şi scalară. În fig. 5.3 (a) se prezintă un exemplu de comparaţie pentru problema adunării unei mulţimi de numere.

(*vii*) *Variaţia lui* r_∞, $n_{1/2}$

Formularea metodei de analiză a algoritmilor $n_{1/2}$, prezentată anterior, se bazează pentru simplitate pe presupunerea că cei doi parametri $(r_\infty, n_{1/2})$ pot fi consideraţi constanţi şi, deci, pot fi scoşi în factori în ecuaţia (5.3). Această presupunere nu se verifică întotdeauna practic, deoarece, aşa cum am arătat în capitolul 2, cei doi parametri au pentru cazuri diferite (cum sînt operaţiile cu registre sau memorie, vectori memoraţi în locaţii succesive sau nu, operaţii cu doi sau trei operanzi etc.) valori diferite. Dacă variaţiile $(r_\infty, n_{1/2})$ sînt mari pentru aceste cazuri diferite, suma din ecuaţia (5.3) poate fi împărţită în mai multe subsume, cîte una pentru fiecare tip de operaţie şi, astfel, se pot folosi valori diferite ale parametrilor. Astfel, se pot folosi valori medii $(\bar{r}_\infty, \bar{n}_{1/2})$ deduse cu

$$\bar{r}_\infty = \left(\Sigma_k \frac{(s_k/s)}{r_{\infty,k}}\right)^{-1} \qquad \bar{n}_{1/2} = \bar{r}_\infty \Sigma_k \left(\frac{n_{1/2,k}}{r_{\infty,k}}\right)\left(\frac{q_k}{q}\right) \qquad (5.8\ a)$$

unde $(r_{\infty,k}, n_{1/2,k})$ sînt parametrii pentru operaţia aritmetică de tip k, iar s_k şi q_k sînt numerele de calcul pentru operaţia de tip k. Timpul de execuţie al algoritmului este

$$T = \bar{r}_\infty^{-1}(s + \bar{n}_{1/2} \cdot q) \qquad (5.8\ b)$$

unde $s = \Sigma_k s_k$ şi $q = \Sigma_k q_k$ sînt numerele totale de calcule.

Calcularea valorii medii este similară cu calcularea performanţei medii a unui calculator, folosind tipuri diferite de instrucţiuni (de exemplu, pachetul Gibson sau Whetstone). (s_k/s) şi (q_k/q) sînt fracţia din calculele aritmetice, respectiv operaţii vectoriale ale operaţiei k. Important este că aceste rapoarte sînt relativ independente de dimensiunea probemei n şi analiza poate continua cu $\bar{n}_{1/2}$ în loc de $n_{1/2}$. În cele mai multe cazuri va fi indicat să se considere r_∞ şi $n_{1/2}$ constanţi şi să se interpreteze diagramele de fază algoritmică pentru gama de valori ale parametrilor care intervin.

5.1.7 Metoda $s_{1/2}$ de analiză a algoritmilor MIMD

Extinderea metodologiei anterioare la calculul MIMD cu fluxuri multiple de instrucţiuni impune definirea unei metode de calcul destul de diferită, pentru care se foloseşte conceptul de segment de lucru prezentat în

§ 1.3.6 (Hockney 1984a, 1985a, 1985c, Hockney și Snelling 1984). Un segment de lucru este o secțiune de program în care activitățile pot fi împărțite între mai multe fluxuri de instrucțiuni logic independente (care nu comunică). Aceste fluxuri pot fi interpretate ca *procese* separate de către calculatorul Denelcor **HEP** cu un singur **PEM**, sau se pot executa de către *procesoare* separate într-o arhitectură multiprocesor, sau **HEP** cu mai multe **PEM**. Din punct de vedere al modelului de calcul, toate aceste situații se tratează în cadrul aceleiași analize, ca fluxuri separate de instrucțiuni. Esența segmentului de lucru este că se sincronizează calculele înaintea și după fiecare segment. Deci, toate operațiile unui segment trebuie terminate înainte să înceapă execuția următorului. Timpul t_i de execuția al unui segment este suma timpului de execuție al celui mai lung flux de instrucțiuni plus timpul de sincronizare al fluxurilor multiple, cu alte cuvinte din ecuația (1.16):

$$t_i = r_\infty^{-1}(s_i/E_i + s_{1/2}) \tag{5.9}$$

unde s_i este numărul operațiilor în virgulă mobilă între perechi de numere în segmentul de lucru i, pe care îl denumim în continuare cantitatea de muncă din segment, sau granularitatea segmentului; E_i este eficiența, E_p, a utilizării procesului în segmentul de lucru i (indicele p este eliminat); r_∞ este performanța asimptotică (sau maximă) în Mflop/s; și $s_{1/2}$ este overhead-ul de sincronizare măsurat în operații în virgulă mobilă echivalente.

Atît valoarea lui r_∞ cît și a lui $s_{1/2}$ din ecuația (5.9) depind de numărul fluxurilor de instrucțiuni sau procesoare folosite. De exemplu, dacă sînt p procesoare, r_∞ din ecuația (5.9) va fi de p ori performanța asimptotică a unui singur procesor. Și valoarea lui $s_{1/2}$ depinde de tipul sincronizării și eficiența instrumentelor software pentru sincronizare. Valorile măsurate ale lui $(r_\infty, s_{1/2})$ pentru cazuri diferite cînd execuția se realizează pe **CRAY X-MP** sînt prezentate în § 2.2.6 (Hockney 1985a), pentru Denelcor HEP în capitolul 3 (Hockney 1984a, 1985c, Hockney și Snelling 1984), iar pentru **FPS-5000** în lucrarea lui Curington și Hockney (1986).

Avînd stabilit timpul de execuție pentru un segment de lucru, putem lua în considerare timpul de execuție al unui program MIMD. În orice program MIMD, va fi o cale critică, al cărei timp de execuție va fi timpul de execuție al întregului program. În unele cazuri calea critică este evidentă (poate fi o singură cale), dar în altele poate fi foarte dificil să se determine, cînd, de exemplu, depinde de date și rămîne necunoscută pînă în momentul execuției. Totuși, pentru a continua analiza, trebuie să presupunem că traseul critic este cunoscut. Timpul de execuție T al unui algoritm MIMD se calculează prin sumarea ecuației (5.9) pentru fiecare segment de lucru de-a lungul căii critice,

$$T = r_\infty^{-1}[(s/\bar{E}) + s_{1/2} \cdot q] \tag{5.10}$$

unde q este numărul segmentelor de lucru de-a lungul căii critice a algoritmului, $s = \sum_{i=1}^{q} s_i$ este cantitatea de operații de-a lungul căii critice, iar $\bar{E}=$

$= s / \left(\sum_{i=1}^{q} s_i / E_i \right)$ este eficienţa medie a utilizării procesului de-a lungul căii critice a algoritmului.

Pentru algoritmii care corespund acestui model de calcul, observăm că mediul de programare (hardware, software şi posibilităţile compilatorului) este descris de perechea de parametri $(r_\infty, s_{1/2})$, în timp ce algoritmul propriu-zis este caracterizat de tripletul $(s, q, \bar{E})$. Ca şi în cazul calculului SIMD, r_∞ dispare la compararea performanţelor a doi algoritmi MIMD executaţi pe acelaşi calculator. Astfel, $s_{1/2}$ rămîne parametrul calculatorului care determină alegerea celui mai bun algoritm MIMD.

Comparînd ecuaţiile (5.9) şi (5.10) cu (5.2) şi (5.4 a), se observă analogia dintre calculele SIMD şi MIMD. Granularitatea unui segment s, sau mai corect $s/\bar{E}_p$, este analoagă cu lungimea vectorului n; overhead-ul de sincronizare, măsurat de $s_{1/2}$, este analog cu lungimea performanţei jumătate, $n_{1/2}$. La fel, se pot compara algoritmii MIMD prin folosirea diagramelor de fază algoritmice.

În §1.3.6 se arată, şi este evident conform analogiei de mai sus, că dimensiunea granularităţii unui segment de lucru trebuie să depăşească $s_{1/2}$ dacă se doreşte ca viteza de execuţie medie a segmentului să treacă de 50% din cea maximă (presupunem $E_p \approx 1$).

Deoarece în sistemele MIMD costul comunicaţiei între fluxurile de instrucţiuni este adesea mare, valoarea lui $s_{1/2}$ tinde să fie mare, adesea mai multe sute sau mii de operaţii in virgulă mobilă. De aceea este important să împărţim un algoritm MIMD în blocuri de cod mari, independente alocate unor fluxuri diferite de instrucţiuni, ceea ce conduce la valori mari pentru s. Această operaţie se poate realiza cel mai bine prin execuţia paralelă a diferitelor instanţe ale buclelor **DO** cele mai exterioare de unde şi denumirea de paralelism la nivelul cel mai exterior al unui program. Acest mod de programare este opus modului SIMD, unde buclele **DO** cele mai interioare ale programului se paralelizează prin înlocuirea cu instrucţiuni vectoriale. Din păcate, paralelizarea la nivelul cel mai exterior solicită o cunoaştere de ansamblu a structurii programului, şi este mult mai dificil de automatizat decît vectorizarea buclelor **DO** simple sau duble, care se pot executa în urma unei examinări locale a programului. În consecinţă, paralelizarea unui program pentru calculul MIMD solicită probabil mult mai mult inetrvenţia programatorului, decît vectorizarea unui program pentru calculul SIMD, şi poate impune uneori restructurarea întregului program.

5.2. Recurenţe

O recurenţă este o secvenţă de evaluări în care valoarea ultimului termen din secvenţă depinde de unul sau mai mulţi din termenii calculaţi anterior. Recurenţele intervin deseori în analiza numerică: în rezolvarea ecuaţiilor liniare prin eliminarea Gauss; în toate operaţiile cu matrici care solicită produsul interior al vectorilor; în toate metodele iterative, deoarece o aproximare mai bună a unei soluţii se calculează pe baza aproximărilor anterioare; în toate soluţiile ecuaţiilor diferenţiale ordinare în timp,

deoarece valorile la un moment dat depind de cele găsite la momente anterioare; şi în metodele pentru rezolvarea ecuaţiilor diferenţiale în spaţiu.

Evaluarea unei recurenţe reprezintă o problemă specială pentru un calculator paralel deoarece chiar definiţia este dată în termenii unei evaluări secvenţiale, şi ar rezulta că numai un termen poate fi evaluat la un moment dat, nerezultînd nici un orizont pentru evaluarea paralelă. De prisos să spunem că problema poate fi reformulată (la costul unor operaţii aritemtice suplimentare) astfel încît să permită evaluarea paralelă. Vom exemplifica această soluţie în primul rînd în termenii problemei simple a evaluării sumei unei secvenţe de numere (§5.2.1 şi §5.2.2). În ultima parte (§5.2.3) tratăm despre introducerea paralelismului în — se mai poate spune vectorizarea — recurenţa generală, liniară de prim ordin. Algoritmi paraleli pentru rezolvarea recurenţelor au mai descris Kogge şi Stone (1973) şi Ladner şi Fisher (1980).

5.2.1. Suma secvenţială

Recurenţa generală liniară de prim ordin poate fi exprimată ca evaluare a secvenţei x_j cu relaţia de recurenţă :

$$x_j = a_j * x_{j-1} + d_j \text{ pentru } j = 1, 2, \ldots, n \qquad (5.11a)$$

cu valorile cunoscute $x_0, a_1, \ldots, a_n$ şi $d_1, \ldots\ldots, d_n$. Dacă presupunem că $x_0 = a_1 = 0$ nu se pierde generalitatea şi apare un avantaj considerabil. Presupunerea poate fi realizată dacă redefinim d_1 ca $d_1 + a_1 * x_0$, ceea ce vom considera în continuare.

Ca un caz particular al celui anterior ($a_j = 1$) vom considera în primul rînd evaluarea sumei parţiale x_j definită cu :

$$x_j = \sum_{k=1}^{j} d_k \text{ pentru } j = 1, 2, \ldots, n \qquad (5.11b)$$

unde x_j este suma primelor j numere din secvenţa $d_1, \ldots, d_n$.

Sumele parţiale pot fi evaluate uşor cu recurenţa :

$$x_j = x_{j-1} + d_j \text{ pentru } j = 1, 2, \ldots, n \qquad (5.11c)$$

Această metodă secvenţială de evaluare poate fi realizată cu (n — 1) sume de următorul cod **FORTRAN** :

```
      X(i) = D(1)
      DO 1 J = 2, N                    (5.12 a)
    1 X (J) = X(J-1) + D(J)
```

Relaţia dintre modul de memorare a operanzilor, operaţii aritmetice şi timp poate fi prezentată printr-o diagramă, în figura 5.1, pentru n = 8.

Secvenţa evaluărilor are loc din partea inferioară în sens ascendent; operaţiile ce pot fi executate în paralel se prezintă pe acelaşi nivel orizontal. Este clar că la fiecare nivel, corespunzător unui moment de timp, se poate executa numai o operaţie (paralelism=1), şi spunem că algoritmul de sumare secvenţială are

$$n - 1 \text{ sume cu paralelism } 1 \tag{5.12b}$$

iar numărul de operaţii este

$$s = n - 1 \quad q = n - 1 \quad \bar{n} = 1 \tag{5.12c}$$

Pentru a cuantifica transferul, presupunem că în fig. 5.1 axa orizontală indică poziţia relativă a datelor în memoria unui calculator serial sau

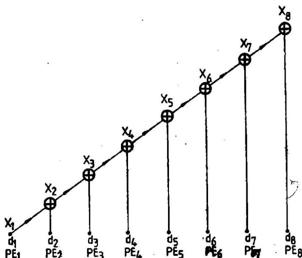


Fig. 5.1 Desfăşurarea în timp a execuţiei metodei de sumare parţială pentru calcularea tuturor sumelor parţiale a 8 numere, $x_j = \sum_{k=1}^{j} d_k$, $j = 1, \ldots, 8$. Axa verticală este timpul, cea orizontală reprezintă numărul locaţiilor de memorie sau a elementelor procesoare. Transferul datelor prin memorie este indicat cu săgeţi.

pipeline, iar în cazul unui masiv de procesoare, numărul acestora. Variabilele din aceeaşi poziţie orizontală nu se suprapun în mod necesar. Dacă nu se cere rescrierea rezultatelor în aceleaşi locaţii de memorie, aceste variabile pot fi memorate în locaţii diferite ale aceluiaşi procesor din masiv, sau într-o secvenţă diferită de locaţii ale unui calculator serial sau pipeline.

O operaţie de transfer paralelă unitară este definită ca o deplasare a tuturor elementelor unui masiv în paralel la o serie de procesoare vecine. În cazul cel mai simplu al topologiei vecinătăţii proxime într-un masiv uni-dimensional de procesoare, aceasta poate însemna deplasarea tuturor

elementelor unui procesor la dreapta sau la stînga. Într-un masiv de procesoare bidimensional, cum este **ICL DAP**, o unitate de transfer poate deplasa toate elementele unei matrice bi-dimensionale, memorată de masivul de procesoare, către procesoarele vecine pe direcția N, S, E sau V. La un masiv de procesoare, după o operație paralelă memorarea poate fi suprimată funcție de starea unui indicator aflat în fiecare procesor. Astfel, numărul elementelor efectiv deplasate după o operație de transfer paralel este sub controlul programului. Numărul elementelor efectiv deplasate (adică utilizarea unei operații de transfer paralel) reprezintă paralelismul operației de transfer.

Analiza figurii 5.1 arată că algoritmul de adunare secvențială necesită un transfer de o unitate la dreapta, la fiecare instanță de timp. În deplasare ete implicat numai un număr (valoarea sumei acumulate curente), deci paralelismul este 1. Rezultatul este confirmat de faptul că există numai o săgeată oblică, ce indică deplasarea unui număr, la fiecare instanță de timp. Prin urmare algoritmul complet necesită

$$n - 1 \text{ operații de transfer cu paralelism } 1. \qquad (5.12d)$$

5.2.2 Adunarea în cascadă

Soluția alternativă paralelă la evaluarea sumelor parțiale poate fi înțeleasă cel mai ușor din diagrama de transfer a algoritmului. Aceasta apare în fig. 5.2(a) pentru cazul $n = 8$, iar algoritmul poartă numele de adunare paralelă în cascadă. Un număr de n registre acumulator sînt încărcate inițial cu operanzii de sumat. La nivelul 1, o copie a conținuturilor acumulatoarelor este deplasată o poziție la dreapta și adunată la conținutul acumulatoarelor pentru a forma suma operanzilor adiacenți. La următorul nivel, procesul se repetă dar cu o deplasare de două poziții la dreapta, producînd prin urmare sume de grupe de cîte 4 elemente. La efectuarea deplasării, prin stînga se introduc zerouri. În general, la momentul l, s-a efectuat o deplasare de 2^l poziții și la nivelul $l = log_2 n$ acumulatoarele conțin sumele parțiale cerute.

Metoda sumării parțiale în cascadă poate fi exprimată într-un limbaj vectorial asemănător **FORTRAN**-ului cu :

```
      X = D
      DO 1 L = 1, LOG2N                    (5.13a)
    1 X = X + SHIFTR (X, 2** (L - 1))
```

unde **X** și **D** sînt vectori cu n elemente, + este o adunare paralelă a n elemente și **SHIFTR (X, L)** este o funcție vectorială care plasează o copie a lui **X**, deplasată L locații de memorie la dreapta în vectorul temporar **SHIFTR**. Elementele vectorului **X** nu sînt modificate. Prin urmare, această metodă necesită

$$\log_2 n \text{ adunări cu paralelism } n \qquad (5.13b)$$

rezultînd numărul de operaţii

$$s = n\log_2 n \quad q = \log_2 n \quad \bar{n} = n \tag{5.13c}$$

Dacă presupunem topologia de conectare a vecinului proxim, sînt necesare 2^{l-1} operaţii de transfer la nivelul l, rezultînd un total de $1 + 2 + + 4 + \ldots + n/2$ sau

$$n - 1 \text{ operaţii de transfer cu paralelism } n \tag{5.13d}$$

Formalismul descris pentru metoda sumării în cascadă, care păstrează lungimea vectorului la valoarea maximă, n, este indicat pentru masive de procesoare, deoarece adunările redundante cu 0 nu consumă timp supli-

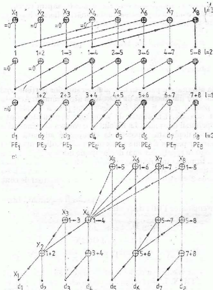


Fig. 5.2(a) Diagrama pentru metoda de adunare paralelă în cascadă cu formarea sumelor parţiale. Odată cu deplasarea vectorului spre dreapta, prin stînga se introduc zerouri. Dacă se cere numai totalul x_8, vor trebui executate numai operaţiile marcate de cercurile pline. (b) Metoda Martin Dates de calcul al sumelor parţiale

mentar. La un calculator vectorial, totuși este necesar mai puțin timp dacă operațiile redundante se omit, iar lungimea vectorului este redusă. În acest caz metoda solicită

$$1 \text{ adunare cu paralelism } n-2^{l-1},$$

$$\text{pentru } l = 1,2,\ldots,\log_2 n \qquad (5.13e)$$

rezultînd

$$s = n\log_2 n - n + 1 \quad q = \log_2 n \quad \bar{n} \approx n[1 - (\log_2 n)^{-1}] \qquad (5.13f)$$

În cadrul analizei algoritmilor din §5.2.3 vom presupune că lungimea vectorului se reduce în acest mod.

Fiecare operație a acestui algoritm are aproximativ un paralelism n (ce variază de la $n-1$ la $n/2$ dacă lungimea vectorului este redusă). Oricum numărul total de operații aritmetice scalare a crescut de $n - 1$ pentru metoda adunării secvențiale la $n\log_2 n$. Costul creșterii paralelismului de la 1 la n a fost prin urmare creșterea substanțială a numărului operațiilor aritmetice scalare. De exemplu, pentru $n = 1024$, metoda de adunare parțială în cascadă solicită un număr de 10 ori mai mare de adunări. La un calculator serial care poate executa numai o adunare la un moment dat, este evident că metoda sumării parțiale în cascadă nu merită să fie folosită. Trebuie să vedem dacă pentru un masiv de procesoare sau un calculator pipeline creșterea de performanță, prin execuția a n operații scalare în paralel, este suficientă pentru a depăși creșterea numărului total de operații (vezi § 5.2.3).

Algoritmul adunării parțiale în cascadă se simplifică considerabil în cazul special în care se cere numai rezultatul *sumei totale* (în cazul nostru x_8). În fig. 5.2 am accentuat cu linii îngroșate și cercuri pline acele operații aritmetice și de transfer care contribuie la calculul lui x_8. Trebuie observat că acestea reprezintă numai o fracție mică din operațiile necesare pentru calcularea tuturor sumelor parțiale. Calculele pentru suma totală formează un arbore binar (cu forma unei cascade), în care numărul operațiilor și prin urmare paralelismul este înjumătățit la fiecare nivel l al procesului de calcul. Presupunînd, pentru simplitate, că n este o putere a lui 2, numărul operațiilor pentru metoda sumei totale în cascadă este :

$$1 \text{ adunare cu paralelism } n2^{-l} \text{ pentru } l = 1, 2, \ldots, \log_2 n \qquad (5.14a)$$

Prin urmare, numărul adunărilor scalare este :

$$n/2 + n/4 + \ldots + 2 + 1 = n - 1 \qquad (5.14b)$$

și observăm că acest algoritm are același număr de operații scalare ca și metoda adunării secvențiale, chiar dacă reorganizarea calculului într-un arbore binar a introdus posibilitatea calculului paralel. Această metodă pentru calculul sumei totale este denumită în mod obișnuit adunare în cascadă.

Martin Oates a subliniat (într-o comunicare particulară) că sumele parțiale mai pot fi calculate în paralel cu aproximativ jumătate din numărul operațiilor aritmetice redundante prezente în metoda de adunare parțială în cascadă inițială prezentată în fig. 5.2(a). Varianta Martin Oates

este ilustrată în fig. 5.2(b). Sumele parţiale se calculează ierarhic. Întîi se adună perechile adiacente, apoi acestea se combină pentru a forma sume de 4 numere adiacente, acestea se folosesc în continuare pentru calculul sumelor de 8 numere ş.a.m.d. La fiecare nivel se execută n/2 sume în paralel, iar pentru $\log_2 n$ nivele se obţin

$$s = (n/2) \log_2 n \quad q = \log_2 n \quad \bar{n} = n/2 \tag{5.14c}$$

în comparaţie cu aproximativ $s = n\log_2 n$ sume în cazul metodei iniţiale.

Metoda lui Oates pune o problemă de programare, în sensul că deplasarea sau indexarea de la fiecare nivel nu este la fel de regulată ca în cazul metodei iniţiale. Dacă se foloseşte un calculator vectorial, cum este **CRAY X-MP**, deşi operaţiile de la fiecare nivel sînt independente, ele nu pot fi executate cu o singură operaţie vectorială. Ele vor trebui implementate ca o secvenţă de operaţii „scalar + vectorial", deci nu se atinge întregul potenţial al execuţiei paralele. De exemplu, la ultimul nivel al calculelor din fig. 5.2(b), scalarul x_4 se adună la vectorul format din valorile curente ale ultimelor 4 elemente ale masivului *d*. La nivelul anterior trebuie executate 2 astfel de operaţii în secvenţă, cu vectori de lungime 2, şi aşa mai departe.

Dacă se foloseşte un masiv de procesare, ca ICL DAP, sînt necesare atît operaţii de deplasare, cît şi de mascare, pentru a asigura sumarea corectă a componentelor. În mod evident, circulaţia datelor este mult mai simplă în cazul iniţial. În orice situaţie particulară, trebuie analizate detaliat toate alternativele pentru a deduce dacă reducerea operaţiilor redundante nu creşte prea mult complexitatea programului. Metoda lui Oates este un caz particular al unei clase de algoritmi pentru aşa numitul „calcul prefix paralel" prezentat de Ladner şi Fisher (1980). În continuare vom analiza performanţele metodei iniţiale şi vom lăsa ca exerciţiu calculul efectului utilizării variantei Oates.

5.2.3 Performanţa relativă

În continuare se vor compara performanţele metodelor de adunare în cascadă şi secvenţială, folosind metoda de analiză $n_{1/2}$ (§ 5.1.6). Acest model de calcul presupune că există o memorie infinită şi că nu există conflicte de acces la blocurile de memorie. Deşi nerespectată în practică, presupunerea de mai sus asigură o primă bază pentru alegerea algoritmilor. Un programator responsabil de evaluarea sumelor în cazul unui calculator real trebuie, desigur, să acorde mare atenţie proprietăţilor memoriei.

Vom considera întîi problema calculului sumei totale, pentru care vom compara metoda secvenţială cu cea de adunare în cascadă. Ambele au acelaşi număr de operaţii aritmetice, $(n - 1)$. Totuşi operaţia in cascadă are mai puţine instrucţiuni vectoriale ($log_2 n$) decît metoda secvenţială $(n-1)$. Prin urmare, este evident că dacă se execută ambii algoritmi în unitatea vectorială a unui calculator, întotdeauna metoda în cascadă va fi mai bună, deoarece sînt mai puţine operaţii de iniţializare a operaţiilor vectoriale. Se poate pune întrebarea dacă nu ar fi mai bine ca metoda secvenţială să se execute în unitatea scalară, eliminîndu-se astfel overhead-ul

asociat cu operaţiile vectoriale. Unitatea scalară are o rată asimptotică mai lentă şi de aceea va interveni o dependenţă de dimensiunea problemei. De la o anumită valoare în sus va fi mai bună metoda în cascadă executată de unitatea vectorială, care este caracterizată de o performanţă asimptotică mai înaltă. Pînă la atingerea acelei valori este indicată folosirea unităţii scalare, chiar dacă are o performanţă asimptotică mai scăzută.

Pentru comparaţie se poate folosi ecuaţia (5.7) cu $s^{(s)} = (n-1)$, $s^{(v)} = (n-1)$, $q^{(v)} = \log_2 n$, în timp ce ecuaţia pentru linia de performanţă egală este

$$n_{1/2} = [R_\infty(n-1) - (n-1)]/\log_2 n \qquad (5.15a)$$

sau

$$n_{1/2}/n = (R_\infty - 1)(1 - n^{-1})/\log_2 n \qquad (5.15b)$$

În fig. 5.3(a) se prezintă diagrama de fază algoritmică pentru un raport al vitezelor vectorială şi scalară de $R_\infty = 2, 5, 10, 50, 100$. Iată interpretarea diagramei. Liniile de $n_{1/2}$ constant se află la 45° faţă de axă, şi apare şi linia de $n_{1/2} = 20$ pentru CRAY-1. Calculatorul are $R_\infty \approx 10$, iar intersecţia între linia de $n_{1/2}$ constant şi linia de R_∞ constant furnizează valoarea limită a dimensiunii problemei. În general, concluzia este că adunarea a pînă la 8 numere trebuie efectuată în unitatea scalară (sau cu instrucţiuni

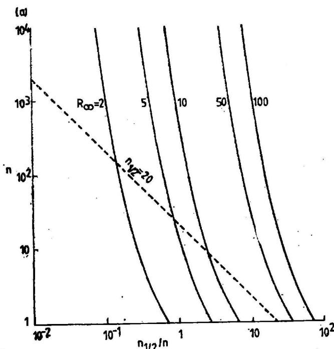


Fig. 5.3(a) Comparaţie între metoda de sumare secvenţială executată în unitatea scalară şi metoda de calcul în cascadă a sumei totale în unitatea vectorială. R_∞ este raportul între vitezele vectorială şi scalară. Linia întreruptă corespunde la $n_{1/2} = 20$.

scalare). în timp ce probleme mai mari trebuie rezolvate de unitatea vectorială. Este evident, şi se observă pe diagramă, că odată cu scăderea raportului vitezelor de execuţie vectorială/scalară, unitatea scalară poate fi folosită pentru probleme cu vectori mai lungi.

Cînd se calculează toate sumele parţiale, se pot considera trei alternative : executarea metodei secvenţiale sau în cascadă în unitatea vectorială sau metoda secvenţială executată în unitatea scalară. Nu trebuie să considerăm şi cazul execuţiei adunării în cascadă de către unitatea scalară, deoarece va fi întotdeauna mai lentă decît metoda secvenţială. Prin urmare, există trei curbe de performanţe egale ce trebuie calculate pentru toate cele trei perechi pasibile de algoritmi.

Pentru a analiza metodele de adunare secvenţială şi în cascadă executate în unitatea vetorială, va trebui să calculăm numărul total de operaţii aritmetice, ca şi numărul de operaţii vectoriale q pentru fiecare din ele. Pentru suma secvenţială :

$$s = n - 1 \quad q = n - 1 \quad \bar{n} = 1 \qquad (5.16\text{ a})$$

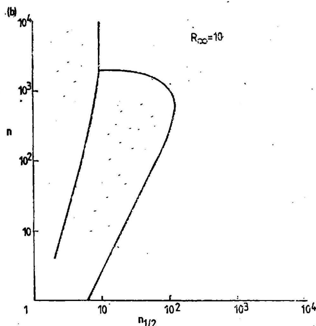


Fig. 5.3(b) Comparaţie între trei algoritmi pentru calculul sumelor parţiale. Se compară metodele secvenţială şi în cascadă executate în unitatea vectorială şi metoda secvenţială executată în unitatea scalară. $R_\infty = 10$.

Pentru suma parţială în cascadă :

$$s = \sum_{l=0}^{\log_2 n - 1} (n - 2^l) = n\log_2 n - (n - 1)$$

$$q = \log_2 n$$

$$\bar{n} = [n\log_2 n - (n - 1)] / \log_2 n \approx n[1 - (1/\log_2 n)] \quad (5.1\text{b})$$

Cu aceste expresii, formula ce definește linia de performanță egală între cele două metode poate fi scrisă imediat, plecînd de la ecuația (5.5):

$$n_{1/2} = [n\log_2 n - 2(n - 1)]/(n - \log_2 n - 1) \qquad (5.16c)$$

Comparația între metoda secvențială executată de unitatea scalară și cele două metode anterioare poate fi realizată plecînd de la expresiile (5.16a, b). Liniile de performanță egală obținute prin substituire în ecuația (5.7 sînt: cu metoda secvențială

$$n_{1/2} = R_\infty - 1 \qquad (5.17a)$$

și cu metoda de adunare parțială în cascadă

$$n_{1/2} = [(R_\infty + 1)(n - 1)/\log_2 n] - 1 \qquad (5.17b)$$

Diagrama de fază algoritmică se obține prin desenarea celor trei curbe (5.16c), (5,17a), (5.17b) în planul ($n_{1/2}$, n). Rezultatul este prezentat în 5.3(b).

Diagrama de fază arată că pentru oricare calculator (definit de o valoare fixă a lui $n_{1/2}$, o linie verticală în fig. 5.3(b)), nici una din metode nu este permanent mai bună. Nu vom putea spune că deoarece folosim un calculator paralel, algoritmul în cascadă proiectat pentru calculul paralel este întotdeauna cel mai bun. Alegerea depinde într-un mod complex de relația dintre dimensiunea problemei n și parametrii $n_{1/2}$ și R_∞ ai calculatorului, care poate fi exprimată în mod adecvat cu o diagramă de fază algoritmică.

Pentru $n_{1/2} < R_\infty - 1$ ($R_\infty = 10$ în fig. 5.3(b)), alegerea algoritmului se face între metoda secvențială și în cascadă executate de unitatea vectorială. În acest caz, pentru orice calculator va exista întotdeauna o dimensiune a problemei peste care metoda secvențială este mai bună, furnizată de poziția curbei de performanța egală. De exemplu, dacă calculatorul are $n_{1/2} = 8$, metoda secvențială va fi mai bună la adunarea a mai mult de 1000 numere. În acest caz, lungimile vectorilor operanzi au devenit atît de mari încît parametrul $n_{1/2}$ al calculatorului este neglijabil ($n/n_{1/2}$ este mare). Sîntem în regimul de calcul serial (vezi ecuația (1.9c) și (5.4c)) și nu este surprinzător că algoritmul proiectat pentru calculul secvențial serial este cel mai bun. Formulat altfel, se poate spune că problema este atît de mare (măsurată prin n) încît paralelismul calculatorului (măsurat prin $n_{1/2}$) este neglijabil, iar calculatorul se comportă în acest caz ca unul serial, chiar dacă are paralelism finit. Variantele seriale ale algoritmilor care minimizează pe s sînt, în aceste condiții, cele mai bune.

Dacă $n_{1/2} > R_\infty - 1$, alegerea se face între metoda în cascadă executată de unitatea vectorială și metoda secvențială executată de unitatea scalară. De exemplu, dacă $n_1/_2 = 20$, probleme cu dimensiunea mai mică ca $n \approx 10$ sau mai mare ca $n \approx 2000$, nu trebuie executate în unitatea scalară. În primul caz, vectorii nu sînt suficient de lungi pentru a justifica overhead-ul de inițializare, iar în ultimul caz, se execută prea multe operații aritmetice suplimentare pentru a face metoda în cascadă mai avanta-

joasă. În regiunea intermediară, $10 \leq n \leq 2000$, este indicată utilizarea unității vectoriale cu performanța ei asimptotică mai mare. Totuși, pentru $n_{1/2} \approx 120$ (în cazul $R_\infty = 10$), overhead-ul de inițializare este prea mare pentru a fi compensat de performanța vectorială mai mare, și metoda de adunare secvențală executată în unitatea scalară este întotdeauna cea mai bună. Fig. 5.3 (b) poate fi desenată pentru alte valori ale lui R_∞, ca în fig. 5.3 (a). Granița între zonele de utilizare a unității scalare, și vectorială se va deplasa spre dreapta la creșterea lui R_∞. Curba de performanță egală între cei doi algoritmi executați în unitatea vectorială nu se va modifica

Dacă problema sumei parțiale s-ar executa pe paracalculator, sau un masiv de procesoare finit cu mai multe procesoare decît numărul operanzilor, considerăm că $n_{1/2} = \infty$. În acest caz, timpul de execuție este proporțional cu numărul operațiilor vectoriale, q. În consecință, metoda în cascadă este întotdeauna cea mai bună, deoarece are numai $log_2 n$ operații vectoriale, în comparație cu $(n - 1)$ pentru metoda secvențială. Dacă se alege un algoritm pentru un masiv de procesoare, trebuie considerată influența operațiilor de transfer necesare. Numărul acestor operații este același pentru ambele metode ($n - 1$, conform ecuațiilor (5.12d) și (5.13d)). Deci, includerea acestui timp nu influențează alegerea algoritmului. Funcție de timpii de execuție a unei operații de tranfer și a uneia aritmetice, pentru un anumit algoritm, timpul consumat pentru transferuri poate fi o componentă importantă (sau chiar dominantă) a timpului total de execuție. Dacă transferul este de γ ori mai rapid decît o operație aritmetică, adică

$$\gamma = \frac{\text{timpul pentru execuția unei operații aritmetice paralelă}}{\text{timpul pentru execuția unei operații de transfer}}$$

atunci raportul între timpul consumat pentru transferuri și timpul consumat pentru operații aritmetice este, în cazul metodei de adunare în cascadă :

$$t_R/t_A = \gamma^{-1}[(n - 1)/\log_2 n] \tag{5.19 a}$$

de unde deducem că transferurile domină operațiile aritmetice dacă

$$\begin{aligned} n &> 64 \quad \text{dacă } \gamma \simeq 10 \\ n &> 1024 \quad \text{dacă } \gamma \simeq 100 \end{aligned} \tag{5.19 b}$$

Deci, operațiile de transfer le vor domina pe cele aritmetice pentru vectori mai lungi ca n_3, unde

$$n_3 = 1 + \gamma \log_2 n_3 \tag{5.19 c}$$

Deoarece $\gamma = 10$ este o valoare tipică, această situație caracterizează toate problemele, mai puțin cele banale. Cu alte cuvinte, viteza de execuție a operațiilor aritmetice nu trebuie crescută, fără o reducere corespunzătoare a timpului de execuție al transferurilor.

O modalitate de reducere a timpului de transfer este de a asigura conexiuni între procesoarele aflate la distanţă mare. În cadrul analizei anterioare am presupus pentru simplitate un masiv uni-dimensional cu conexiuni între vecinii cei mai apropiaţi. Cele mai multe masive de procesoare sînt bi- sau multi-dimensionale. De exemplu, **ICL DAP** este un masiv bi-dimensional de 64×64 procesoare. Dacă acestea sînt interpretate ca un vector unic cu 4096 elemente scrise linie cu linie în masiv, atunci o operaţie de transfer între liniile vecine rezultă într-o deplasare de 64 poziţii. Cazul general al deplasării datelor într-un masiv k-dimensional este discutat în § 5.5.5 şi în lucrările lui Jesshope (1980a, b, c). Alte modalităţi de interconectare, ca amestecul perfect, oferă conexiuni la mare distanţă (vezi § 3.3.4 şi § 3.3.5). Cititorul este invitat să calculeze efectul unor astfel de conexiuni asupra comparaţiilor anterioare.

5.2.4 Reducerea ciclică

Recurenţa lineară generală de prim ordin (ecuaţia 5.11a) poate fi evaluată secvenţial plecînd de la definiţia recurenţei cu următorul cod FORTRAN:

```
      X(1) = A(1) * X(0) + D(1)
      DO 1 J = 2, N                          (5.20)
    1 X(J) = A(J) * X(J - 1) + D(J)
```

Sînt necesare

2n operaţii aritmetice cu paralelism 1, (5.21 a)

şi

n operaţii de transfer cu paralelism 1 (5.21 b)

Diagrama de transfer pentru algoritmul secvenţial este prezentată în fig. 5.4. Pentru simplitate, nu vom număra separat diferitele tipuri de operaţii aritmetice, deşi pot avea timpi de execuţie diferiţi. Raportul între timpul de execuţie al unei înmulţiri şi al unei adunări depăşeşte rareori 2 şi este deseori aproape de 1. În particular, pentru un calculator pipeline atît pipeline-urile pentru adunare cît şi cele pentru înmulţire, cînd sînt complet utilizate, furnizează un rezultat la fiecare perioadă a orologiului. Pentru $n > n_{1/2}$ timpul mediu pentru o adunare sau înmulţire este aproximativ acelaşi.

Algoritmul paralel echivalent metodei de adunare în cascadă este cunoscut sub denumirea de *reducere ciclică* şi are o largă aplicabilitate în analiza numerică, în particular cînd se încearcă introducerea paralelismului într-o problemă. Îl vom folosi, de exemplu, din nou pentru rezolvarea sistemelor de ecuaţii tridiagonale algebrice într-o manieră paralelă (vezi § 5.4.3). Recurenţa originală (ecuaţia 5.11a) leagă termenii vecini din

secvență, adică x_j cu x_{j-1}. Ideea de bază pentru reducerea ciclică este combinarea termenilor adiacenți ai recurenței împreună, într-o astfel de manieră încît să se obțină o relație între oricare alt termen din secvență, adică să se stabilească o relație între x_j și x_{j-2}. Se găsește tot o relație de recurență lineară de prim ordin, deși coeficienții sînt diferiți și relația se stabilește între termeni alternativi. În consecință procesul poate fi repetat (într-o manieră ciclică) pentru a obține recurențe ce leagă fiecare al patrulea termen, fiecare al optulea și așa mai departe. Cînd recurența implică fiecare n termeni (de exemplu, după log_2n nivele de reducere), valoarea la fiecare punct din secvență este legată numai de valori din afara domeniului care sînt cunoscute sau nule, deci soluția a fost găsită. Cînd metoda este folosită pe calculatoare seriale, numărul ecuațiilor recurente care sînt folosite este redus la jumătate la fiecare nivel succesiv, de unde și termenul „reducere ciclică". Pe un calculator paralel sîntem interesați în păstrarea unui paralelism înalt și, de fapt, nu vom reduce numărul ecuațiilor care sînt folosite la fiecare nivel, Prin urmare, numele algoritmului este într-un anumit fel echivoc.

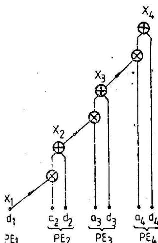


Fig. 5.4 Diagrama de transfer pentru evaluarea secvențială a recurenței generale de prim ordin. În acest caz, n = 4. Variabilele marcate cu aceeași acoladă se află în același PE. Un PE este folosit pentru evaluarea fiecărui termen al recurenței.

Vom deduce acum algebra metodei de reducere ciclică. Să scriem din nou relația de recurență originală pentru 2 termeni succesivi :

$$x_j = a_j x_{j-1} + d_j \tag{5.22a}$$

și

$$x_{j-1} = a_{j-1} x_{j-2} + d_{j-1} \tag{5.22b}$$

Introducînd ecuația (5.22b) în ecuația (5.22a) obținem :

$$x_j = a_j a_{j-1}\, x_{j-2} + a_j d_{j-1} + d_j \tag{5.23a}$$

$$= a_j^{(1)} x_{j-2} + d_j^{(1)} \tag{5.23b}$$

unde ecuația (5.23 b) este o recurență liniară de prim ordin între termeni alternativi ai secvenței cu un nou set de coeficienți definiți cu :

$$a_j^{(1)} = a_j a_{j-1}, \quad d_j^{(1)} = a_j d_{j-1} + d_j \tag{5.23c}$$

Aplicarea repetată a procesului de mai sus poate fi pe scurt prezentată prin ecuațiile reduse pentru nivelul 1. Exponenții notează numărul nivelului.

$$x_j = a_j^{(l)} x_{j-2^l} + d_j^{(l)} \begin{cases} l = 0, 1, \ldots, \log_2 n \\ j = 1, 2, \ldots, n \end{cases} \tag{5.24a}$$

unde

$$a_j^{(l)} = a_j^{(l-1)} a_{j-2^{l-1}}^{(l-1)} \qquad (5.24b)$$

$$d_j^{(l)} = a_j^{(l-1)} d_{j-2^{l-1}}^{(l-1)} + d_j^{(l-1)} \qquad (5.24c)$$

şi iniţial

$$a_j^{(0)} = a_j, \quad d_j^{(0)} = d_j \qquad (5.24d)$$

Dacă indicele oricărui a_j, d_j sau x_j este în afara domeniului definit $1 \leqslant j \leqslant n$, rezultatul corect se obţine prin adoptarea valorii zero. Cînd $l = \log_2 n$ toate referinţele la $x_{j-2^l} = x_{j-n}$ în ecuaţia (5.24a) sînt în afara domeniului definit, de aici soluţia recurenţei este

$$x_j = d_j^{(\log_2 n)} \qquad (5.24e)$$

Prin urmare metoda generează succesiv coeficienţii $a_j^{(l)}$ şi $d_j^{(l)}$ definiţi cu ecuaţiile (5.24b, c), pînă ce se află $d_j^{(\log_2 n)}$. Aceasta este soluţia pentru relaţia de recurenţă.

La prima vedere poate părea că ecuaţia (5.24c) nu poate fi evaluată în paralel. În definitiv, ecuaţia pentru $d_j^{(l)}$ este aparent aceeaşi cu relaţia de recurenţă secvenţială originală cu x înlocuit de d. Diferenţa fundamentală între ecuaţia (5.24c) şi relaţia de recurenţă originală (5.22a) este că valoarea lui $d_{j-2^{l-1}}^{(l-1)}$ şi $d_j^{(l-1)}$ din membrul drept al ecuaţiei (5.24c) sînt valori bine cunoscute, calculate la nivelul anterior (l—1). Aceşti $d_j^{(l-1)}$ sînt variabile distincte de $d_j^{(l)}$ din membrul stîng. Prin urmare, ultimul ($d_j^{(l)}$; $j = 1, \ldots, n$) poate fi evaluat cu o singură operaţie paralelă sau instrucţiune vectorială. Aceste relaţii sînt clasificate cu diagramele de transfer pentru evaluarea lui $a_j^{(l)}$ (fig. 5.5) şi $d_j^{(l)}$ (fig. 5.6).

În aceste diagrame prezentăm numai valorie pentru $a_j^{(l)}$ şi $d_j^{(l)}$. Observăm că sînt necesare numai aproximativ jumătate din valorile $a_j^{(l)}$. În particular nu este necesară nici una cînd $l = 3$. Cantitatea de paralelism variază de la aproximativ n la început la aproximativ $n/2$ pentru nivelul final. La calculatoarele pipeline care lucrează cu vectori de lungime variabilă, trebuie mărită performanţa pentru a reduce lungimea vectorului la valoarea corectă pentru fiecare nivel. La masive de procesoare cu $n \leqslant N$ sau la paracalculator, pentru care timpul de execuţie nu este afectat de lungimea vectorului, paralelismul poate fi menţinut egal cu n la fiecare nivel prin încărcarea tuturor $a = 0$ pentru $-n/2 \leqslant j \leqslant 1$ şi $d_j = 0$ pentru $-n \leqslant j \leqslant 0$ sau altfel considerînd nule valorile care sînt în afara domeniului.

Evident, este destul de complicat să evaluăm performanţele unui algoritm de reducere ciclică, luînd în considerare reducerea paralelismului la fiecare nivel. Totuşi, putem obţine o margine inferioară presupunînd că paralelismul rămîne n la fiecare nivel şi că toţi $a_j^{(l)}$ se calculează. Paralelismul mediu este $[(n-1) + (n-2) + \ldots + (n-2^r) + \ldots + n/2]/$

$/\log_2 n = n[1 - (1 - n^{-1})/\log_2 n]$, de unde pentru valori mari ale lui n obținem :

$3\log_2 n$ operații aritmetice cu paralelism n

și (5.25)

$2(n - 1)$ operații de transfer cu paralelism n

Lăsăm ca un exercițiu pentru cititor să analizeze, în modul prezentat anterior, metodele de adunare secvențiale și în cascadă considerînd pe deplin reducerea paralelismului la fiecare nivel. Recurența generală de prim ordin de mai sus devine suma în cascadă pentru cazul particular $a_j = 1$ pentru $2 \leq j \leq n$, care face ca toate înmulțirile din fig. 5.6 și întreaga figură 5.5 redundante.

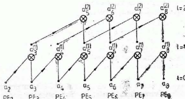


Fig. 5.5 Diagrama de transfer pentru ecuația (5.24 b), calculul paralel al coeficienților a_j în cazul algoritmului de reducere ciclică aplicat recurenței liniare, generale de prim ordin pentru cazul n=8.

Algoritmul de reducere ciclică poate fi implementat într-o formă vectorială prin codul **FORTRAN** :

```
X = D
DO 1 L = 1, LOG 2N
X = A*SHIFTR (X, 2**(L − 1) (+ X          (5.26a)
1 A = A*SHIFTR(A, 2**(L − 1))             (5.26b)
```

unde **X**, **D** și **A** au fost declarate ca vectori. Observăm că, în implementarea acestui cod, adresele din memorie centrală a vectorului **X** și vectorului **SHIFTR**, necesari amîndoi pentru evaluarea expresiei (5.26a), sînt separate prin blocuri de memorie puteri ale lui 2. Aceeași situație este și pentru vectorul **A** în expresia (5.26b). Prin urmare este probabil că conflictele de acces la memorie (vezi § 5.1.5) sînt un impediment serios la evaluarea rapidă a algoritmului de reducere ciclică pentru calculatoarele seriale și pipeline cu un număr de blocuri de memorie egal cu o putere a lui 2. Deoarece Burroughs BSP are un număr prim de blocuri (17), nu este afectat de acest inconvenient (vezi p. § 3.3.8).

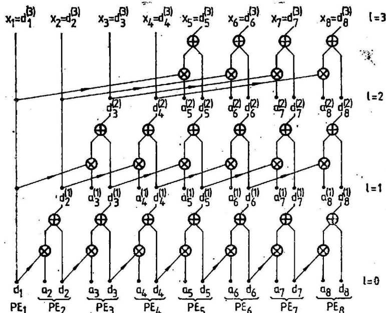


Fig. 5.6 Diagrama de transfer pentru ecuația (5.24 c), calculul paralel al coeficienților $d_j^{(l)}$ în algoritmul de reducere ciclică aplicat recurenței liniare generale de prim ordin. Valorile a_j rezultă în urma calculelor din fig. 5.5, care au loc în paralel cu operațiile descrise de această figură.

5.3 Înmulțirea matricelor

Înmulțirea matricelor este cel mai simplu exemplu de operații cu matrici și ilustrează suficient de bine modurile diferite în care trebuie restructurat un algoritm simplu pentru a corespunde arhitecturii calculatorului pe care urmează să fie executat. Elementele $C_{i,j}$ ale matricei produs ce se calculează din elementele $A_{i,j}$ și $B_{i,j}$ ale matricelor ce se înmulțesc cu formula :

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} \quad 1 \leqslant i,j \leqslant n, \tag{5.27}$$

unde primul indice este numărul liniei, iar al doilea numărul coloanei.

5.3.1 Metoda produsului interior

În mod invariabil, matricele se înmulțesc pe calculatoare seriale prin execuția a trei bucle **DO**, folosind codul FORTRAN care traduce direct formula anterioară :

```
      DO 1 I = 1, N                                  (5.28 a)
      DO 1 J = 1, N
      DO 1 K = 1, N                                  (5.28 b)
    1 C (I, J) = C (I, J) + A (I, K) * B (K, J)
```

unde presupunem că toate elementele C(I, J) au fost inţializate cu zero înaintea procesului de calcul. Expresia de atribuire din cod (5.28b) formează *produsul interior* al liniei i a matricei **A** cu coloana j a lui **B**. Acesta este un caz particular de evaluare secvenţială a sumei unui set de numere, discutată în §5.2 (fie $d_k = A_{i,k} B_{k,j}$ în ecuaţia (5.11b), atunci $x_n = C_{i,j}$). În consecinţă avem fie opţiunea evaluării secvenţiale ca în codul (5.28), fie cea a utilizării metodei de adunare în cascadă. Consideraţiile făcute în §5.2 sînt valabile. Unele calculatoare au o instrucţiune pentru „produs interior" (pentru **CYBER 205**, vezi tabelul 2.4) care trebuie luată în considerare.

Oricum, în evaluarea unui produs matriceal este mai mult paralelism inerent decît cel corespunzător evaluării unei singure sume. Înmulţirea matricelor implică evaluarea a n^2 produse interioare, ce pot fi executate cîte n simultan (metoda *produsului intermediar*) sau n^2 în acelaşi timp (metoda *produsului exterior*).

5.3.2 Metoda produsului intermediar

Această metodă se obţine prin schimbarea ordinei buclelor DO din codul (5.28). Dacă introducem bucla corespunzătoare liniilor I, în poziţia cea mai interioară, obţinem cod care calculează produsul interior al tuturor elementelor unei coloane a lui **C** în paralel:

```
      DO 1 J = 1, N                              (5.29 a)
      DO 1 K = 1, N
      DO 1 I = 1, N                              (5.29 b)
    1 C (I, J) = C (I, J) + A (I, K) * B(K, J)
```

Fiecare termen al buclei corespunzătoare lui **I** poate fi evaluat în paralel, astfel că bucla (5.29b) poate fi înlocuită cu o expresie vectorială. Codul poate fi scris:

```
      DO 1 J = 1, N                              (5.30a)
      DO 1 K = 1, N
    1 C( , J) = C ( , J) + A( , K) * B(K, J)     (5.30b)
```

unde **C**(, **J**) şi **A**(, **K**) sînt vectorii compuşi din coloanele **J** şi **K** ale matricelor **C** şi **A**. Adunarea, +, este o operaţie paralelă cu n elemente, iar înmulţirea, ×, este o operaţie între un scalar B(K, J) cu un vector **A**(, **K**). Prin urmare, paralelismul codului de înmulţire intermediară este n, în comparaţie cu i al metodei de înmulţire interioară. Simpla schimbare a ordinii buclelor **DO** a determinat acest rezultat. Observăm că puteam deplasa bucla **J** spre poziţia mediană, ceea ce ar fi determinat calcularea tuturor produselor interioare ale unei linii în paralel. De obicei elementele coloanelor unei matrice se memorează în locaţii de memorie adiacente (modul **FORTRAN**), în consecinţă conflictele de acces la memorie sînt reduse dacă operaţiile vectoriale au loc între vectori coloană. De aceea se va prefera codul (5.30).

Metoda produsului intermediar, programată în limbaj de asamblare, este cea mai bună pe calculatorul **CRAY-1**. Se obține performanța supervectorială de 138 Mflop/s. Consecința este că, în medie, se vor executa două operații aritmetice într-o perioadă de ceas (o operație executată într-o perioadă de ceas este echivalentă cu 80 Mflop/s). Rezultatul este posibil deoarece operațiile de înmulțire și adunare din expresia (5.30 b) pot fi înlănțuite sub forma unei singure operații compuse executată — în pipeline, ce produce un element al vectorului rezultat **C**(, **J**) la fiecare tact.

Este interesant de observat că metoda produsului intermediar are o performanță superioară metodei produsului interior, chiar pentru calculatoare ca **CDC 7600** care nu au instrucțiuni vectoriale explicite, și nu sînt clasificate ca fiind calculatoare paralele. Totuși **CDC 7600** are unități aritmetice pipeline, iar performanța lor crește cînd operațiile aritmetice se execută într-o manieră regulară, ca în codul serial (5.29 b) pentru operații vectoriale. Această idee a fost exploatată sub forma unor rutine scrise în limbaj de asamblare, optimizate cu multă atenție , ce formează pachetul **STACKLIB** (vezi § 2.3.5) scris la Laboratorul Lawrence Livermore. Aceste rutine execută diverse operații vectoriale diadice sau triadice, ca de exemplu expresia „vector + scalar * vector" din codul (5.30 b), ceea ce conduce la o creștere a performanței cu un factor de 2 față de codul de evaluare serială (5.28 b).

5.3.3. Metoda produsului exterior

Această metodă se obține prin deplasarea buclei corespunzătoare lui **K** din codul (5.28) spre exterior:

```
      DO 1 K = 1, N                                   (5.31 a)
      DO 1 I   1, N
      DO 1 J = 1, N
    1 C (I, J) = C(I, J) + A (I, K) * B (K, J)        (5.31 b)
```

Codul (5.31 b) poate fi înlocuit cu o singură expresie de masiv, în care un termen al produsului interior este evaluat în paralel pentru toate cele n^2 elemente ale lui **C**. Folosind notația introdusă în cap. 4, **C** simbolizează întregul masiv și vom scrie:

```
      DO 1 K = 1 N
    1 C = C + A( , K) * B (K,  )                      (5.32)
```

unde operația de înmulțire este o operație de înmulțire element cu element a unei matrice $n \times n$ obținută prin duplicarea coloanei **K** a matricei **A** și o matrice $n \times n$ obținută prin duplicarea liniei **K** a matricei **B**. Operația de adunare este o adunare element cu element a $n \times n$ elemente. Iată expresia în **DAP FORTRAN**:

```
      DO 1 K = 1, N
    1 C = C + (MATC (A(  ,K)* MAT R(B,(K,   ))        (5.33)
```

unde funcţia **MATC(X)** produce o matrice ale cărei coloane sînt vectorul **X**. Similar, **MATR (X)** produce o matrice ale cărei linii sînt vectorul **X**. Sînt necesare două funcţii deoarece altfel transformarea unui vector într-o matrice ar fi ambiguă (vezi § 4.3.1 (iii) şi (iv).

Este clar că metoda produsului exterior este adecvată pentru un masiv de procesoare cu aceleaşi dimensiuni ca şi matricele (de exemplu, produsul unor matrice 64×64 pe **ICL DAP 64×64**). În acest caz paralelismul structurii hardware este identic cu cel al algoritmului. Deoarece paralelismul a crescut de la n la n^2 în comparaţie cu metoda produsului intermediar, este probabil de asemenea ca metoda produsului exterior să fie superioară pe calculatoare pipeline cu $n_{1/2}$ mari. Pentru matrici 64×64, lungimea vectorului creşte de la 64, în cazul produselor intermediare, la 4096 pentru produsele exterioare. Raportul dintre performanţa metodei produsului exterior, P_o, şi performanţa metodei produsului intermediar, P_m, constă în raportul dintre timpul necesar execuţiei a n operaţii vectoriale de lungime n şi timpul necesar execuţiei unei operaţii vectoriale de lungime n^2 :

$$\frac{P_o}{P_m} = \frac{b(n + n_{1/2})}{n^2 + n_{12}} = \frac{1 + n_{1/2}/n}{1 + n_{1/2}/n^2} \tag{5.34 a}$$

$$\approx \begin{cases} 2 & \text{pentru } n_{1/2} = n > 1 \\ n_{1/2}/n & \text{pentru } n^2 > n_{1/2} > n \end{cases}$$

Pentru cazul $n = 64$ avem

$$\frac{P_o}{P_m} = \begin{cases} 1{,}12 & \text{pentru } n_{1/2} = 8 \\ 2{,}5 & \text{pentru } n_{1/2} = 100 \\ 13 & \text{pentru } n_{1/2} = 1000 \end{cases} \tag{5.34 b}$$

Este clar că se poate alege prea puţin între aceste metode din acest punct de vedere în cazul unui calculator ca **CRAY-1** ce are $n_{1/2}$ mic. Produsul intermediar este favorizat de alte considerente, ca abilitatea de a lucra cu registre vectoriale şi de a folosi înlănţuirea. Totuşi, produsul exterior prezintă avantaje evidente pentru calculatoare pipeline în cazul în care $n^2 > n_{1/2} > n$.

5.3.4 Folosind paralelism n^3

Jesshope şi Craigie (1980) au utilizat o combinaţie interesantă între tehnicile discutate în contextul înmulţirii matricelor pe masive de procesoare al căror ordin diferă de cel al matricelor (de asemenea în Jesshope şi Hockney 1979). De exemplu, cum să se calculeze produsul a două matrice 16×16 pe un sistem **ICL DAP 64×64**? Evident, adoptarea metodei produsului exterior ar fi inoportună deoarece s-ar folosi numai 1/16 din procesoarele disponibile.

Jesshope şi Craigie au observat că în cazul înmulţirii matricelor $n \times n$ se execută n^3 înmulţiri (n^2 produse interioare, fiecare a n înmulţiri [vezi ecuaţia (5.27)] şi că toate aceste produse pot fi evaluate simultan cu un paralelism n^3. Sumarea a n termeni corespunzători tuturor celor n^2 produse interioare se poate executa în $\log_2 n$ paşi, cu paralelism n^3, folosind metoda de adunare în cascadă. Codul FORTRAN echivalent este:

```
      DO 1 I = 1, N
      DO 1 J = 1, N                                   (5.35 a)
      DO 1 K = 1, N
1     C (I, J, K) = A(I, J, K) * B(I, J, K)

      DO 2 L = 1,LOG2N
      K1 = 2** (L-1)
      DO 2 K = 1, N-K1
      DO 2 J = 1, N
      DO 2 I = 1, N
2     (C(I, J, K) = C(I, J, K) + C(I, J, K+K1)        (5.35 b)
```

Mai sus bucla (5.35 a) realizează toate cele n^3 înmulţiri, iar bucla (5.35b) evaluează suma în cascadă. După execuţia celor 4 bucle înlănţuite (5.35 b) elementul C(I, J) al produsului se va găsi în locaţia C(I, J, 1). Codul de mai sus poate fi scris succint, folosind construcţiile paralele introduse în § 4.3.1 (vezi p. 400—1) cu expresia:

$$C = SUM (XPND (A, 3, N) * XPND (B, 1, N), 2) \qquad (5.35 c)$$

În cazul unor matrice 16×16, şi al sistemului ICL DAP 64×64, $n = 16$ şi $n^3 = 4096$ de unde se vede că paralelismul algoritmului este identic cu paralelismul maşinii. Cei trei indici ai matricelor din codul (5.35) sînt transformaţi în cei doi indici ai masivului de procesoare conform unei metode de asignare. Pentru a reduce operaţiile de transfer în timpul sumării în cascadă, trebuie ca valorile cu aceiaşi indici I şi J să se memoreze în procesoare vecine. O astfel de asignare este:

$$C(I, J, K) \text{ memorat în } P(44I + K/4, 4J + MOD (K, 4)), \qquad (5.36)$$

unde indicii procesorului reprezintă numărul liniei, respectiv coloanei, din masiv, ce vor fi calculaţi cu instrucţiuni FORTRAN. Rezultatul acestei asignări este că cele 16 valori cu aceiaşi primi doi indici se vor memora compact într-un masiv de procesoare 4×4, astfel, transferul fiind menţinut la un minim. Operaţiile de transfer la mare distanţă, impuse de extinderea matricelor **A** şi **B** pot fi executate efectiv pe **ICL DAP** folosind facilitatea broadcast (vezi § 3.4.2). Şi funcţia **SUM** din expresia 5.35 c) poate fi optimizată folosind algoritmi la nivel de bit.

5.4 Sisteme tridiagonale

Sistemele tridiagonale formează o clasă foarte importantă a ecuațiilor algebrice liniare. Ele intervin repetat ca aproximări cu diferențe finite ale ecuațiilor diferențiale cu derivate de ordinul 2 : de exemplu, mișcarea armonică simplă și ecuațiile Helmholtz, Laplace, Poisson și de difuzie. În consecință, metode eficiente pentru rezolvarea acestor ecuații se află la baza a numeroși algoritmi numerici importanți pentru rezolvarea lor; de exemplu, metode iterative ca ADI și SLOR, metodele rapide FACR și Buneman pentru rezolvarea anumitor clase a acestor ecuații. Unele din aceste metode se vor prezenta ulterior în § 5.6. Din păcate, multe din tehnici, cum este eliminarea gausiană, foarte eficiente pe calculatoare seriale sînt algoritmi secvențiali și deci inadecvați pentru calculatoare paralele. În continuare vom analiza problema introducerii paralelismului în aceste metode sau adaptarea unor noi algoritmi, ca reducerea ciclică, care în mod inerent sînt mai paraleli. În dezvoltarea acestor tehnici putem folosi mult din experiența cîștigată în timpul studiului recurențelor în § 5.2.

Rezolvarea sistemelor de ecuații tridiagonale este tratată pe larg în literatură. Stone (1973, 1975) Lambiotte și Voight (1975), Swarztrauber (1979 a, b), Kershaw (1982), Kascic (1984 c), Gentzsch (1984) și Schonauer (1987) prezintă metode pentru execuția pe calculatoare vectoriale. Pe de altă parte, Evans (1980), Kowalik și Kumar (1985) și Wang (1985) prezintă metode adecvate pentru calculatoare MIMD.

5.4.1 Eliminarea gausiană

Sistemul tridiagonal de ecuații algebrice liniare poate fi scris ca :

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & & 0 \\ a_2 & b_2 & c_2 & & & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \\ \\ x_n \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ \\ \\ k_n \end{pmatrix} \quad (5.37)$$

sau vectorial :

$$\mathbf{Ax} = \mathbf{k} \quad (5.38)$$

Algoritmul de eliminare gausiană poate fi enunţat astfel:

i) Eliminare înainte

$$w_1 = c_1/b_1$$

$$w_i = c_i/(b_i - a_i w_{i-1}), \quad i = 2, 3, \ldots, n-1 \tag{5.39 a}$$

$$e_i = w_i/c_i \tag{5.39 b}$$

şi

$$g_1 = k_1/b_1,$$

$$g_i = (k_i - a_i g_{i-1})/(b_i - a_i w_{i-1}), \quad i = 2, 3, \ldots, n \tag{5.39 c}$$

sau

$$g_i = (k_i - a_i g_{i-1})\, e_i \tag{5.39 d}$$

sau

$$g_i = (k_i - a_i g_{i-1})\, w_i/c_i \tag{5.39 e}$$

ii) Substituirea înapoi

$$x_n = g_n,$$

$$x_i = g_i - w_i x_{i+1}, \quad i = (n-1), (n-2), \ldots, 1 \tag{5.39 f}$$

În cadrul etapei de eliminare înainte se calculează doi vectori auxiliari, **w** şi **e**, care sînt funcţii numai de coeficienţii matricei **A**. Aceşti vectori sînt coeficienţii din descompunerea triunghiulară a matricei **A** în produsul dintre o matrice triunghiulară inferioară **L** şi o matrice tringhiulară superioară **U**:

$$\mathbf{A} = \mathbf{LU} \tag{5.40}$$

unde

$$L = \begin{pmatrix} e_1^{-1} & 0 & \cdots & & 0 \\ a_2 & e_2^{-1} & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_n & e_n^{-1} \end{pmatrix} \qquad U = \begin{pmatrix} 1 & w_1 & 0 & \cdots & 0 \\ 0 & 1 & w_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & w_{n-1} \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix}$$

Folosind această descompunere, soluţia ecuaţiei (5.37) poate fi exprimată în două etape, pentru un **k** particular:

$$\mathbf{Lg} = \mathbf{K} \tag{5.41 a}$$

$$\mathbf{Ux} = \mathbf{g} \tag{5.41 b}$$

De aici $Ax = LUx = Lg = k$. Etapa de eliminare înainte d (5.39 d) este reprezentată de ecuația (5.41 a), iar etapa substituirii înapoi de ecuația (5.41 b). În timpul acestui proces este posibil ca g să se scrie peste k, iar x peste g, deci vectorul intermediar g nu impune prezența unui spațiu de memorie suplimentar.

Dacă vectorii auxiliari nu sînt precalculați, vom evalua ecuațiile (5.39 a) și (5.39 c) împreună, urmate de ecuația (5.39 f). Numitorul $(b_i - a_{i-1}w_{i-1})$ se evaluează o singură dată și sînt necesare 8n operații aritmetice scalare. Totuși, dacă trebuie rezolvate mai multe ecuații de aceeași matrice A, dar cu vectori k diferiți, numărul operațiilor poate fi redus la 5n dacă cei doi vectori w și e sînt precalculați și memorați. În acest caz folosim ecuațiile (5.39a) și (5.39 b) la calculele preliminare și ecuațiile (5.39 d) și (5.39 f) pentru găsirea soluțiilor pentru fiecare k. Dacă este precalculat un singur vector, w, se omite ecuația (5.39 d) și se folosesc ecuațiile (5.39 e) și (5.39 f). Numărul operațiilor aritmetice scalare crește la 6n, dar, desigur, spațiul de memorie este salvat. Dacă sistemul tridiagonal rezultă din diferența finită a unei derivate de ordinul 2 pentru un caroiaj regulat, $a_i = c_i = 1$. În acest caz $e_i = w_i$ și algoritmul se simplifică considerabil;

$$w_1 = c_1/b_1,$$

$$w_i = (b_i - w_{i-1})^{-1}, \; i = 2, 3, \ldots n-1; \tag{5.42 a}$$

$$g_1 = k_1/b_1,$$

$$g_i = (k_i - g_{i-1})\, w_i. \; i = 2, 3, \ldots, n; \tag{5.42 b}$$

$$x_n = g_n$$

$$x_i = g_i - w_i x_{i+1}, \; i = n-1, n-2, \ldots, 1 \tag{5.42 c}$$

Acum numărul operațiilor scalare s-a redus la 6n fără calcule prealabile și 4 n cu calcularea prealabilă a vectorului w.

Cele trei bucle (5.39 a, c, f) sau (5.42) ale algoritmului de eliminare gausiană sînt toate recurențe secvențiale ce trebuie evaluate cîte un termen la un moment dat. De aici, paralelismul algoritmului este 1. Aceasta, împreună cu faptul că elementele vectorului sînt referite cu incremenți unitari și că numărul operațiilor aritmetice este minimizat, face acest algoritm ideal pentru calculatoarele seriale. De asemenea, absența oricărui paralelism face imposibilă exploatarea caracteristicilor paralele ale unui calculator. În concluzie, algoritmul este inadecvat pentru rezolvarea unui sistem de ecuații tridiagonale pe un calculator paralel.

Totuși, dacă cineva are de rezolvat un număr de m sisteme tridiagonale independente (să luăm $m = 64$), situație frecventă la rezolvarea PDE (vezi § 5.6), atunci eliminarea gausiană va fi cel mai bun algoritm de utilizat pe un calculator paralel. În acest caz, toate cele m sisteme se vor rezolva în paralel prin schimbarea tuturor variabilelor algoritmului în vectori de lungime m. De exemplu, variabila w_i ar deveni vectorul $(w_{i,k};$

$k = 1, \ldots, m$), unde $w_{i,k}$ este evaluarea lui w_i în al k-lea sistem tridiagonal. Cînd se face aceasta, toate operațiile scalare în ecuațiile (5.39) devin operații vectoriale cu vectori de lungime m. Toate expresiile sînt vectorizate și se atinge performanța vectorială maximă. Astfel obținem atît un număr minim de operații aritmetice (prin alegerea procedurii gausiene) cît și paralelismul maxim. Remarcăm că în mod obișnuit aceste două obiective nu pot fi atinse simultan (vezi de ex. § 5.2 pentru recurențe). Dezavantajul principal al acestei metode este că spațiul de memorie necesar crește de m ori în comparație cu rezolvarea unui singur sistem tridiagonal la un moment dat. Alegerea între aceste două alternative se prezintă în continuare în § 5.4.4.

5.4.2 Dublarea recursivă

S-au propus mai multe metode pentru introducerea paralelismului în algoritmul gausian secvențial, printre care remarcăm algoritmul de dublare recursivă al lui Stone (1973). Vom prezenta o variantă a acestui algoritm (Stone 1975). În lucrarea lui Lambiotte și Voight (1975) sînt descrise performanțele unora din acești algoritmi executați pe calculatorul pipeline **CDC STAR 100**. O altă comparație între algoritmi pentru **CDC 7600** și **CDC STAR 100** apare în lucrarea lui Madsen și Rodrique (1976).

O examinare a algoritmului gausian definit de ecuațiile (5.39 a, d, f) sau (5.42) arată că algoritmul constă în 3 recurențe. Ambele recurențe pentru g_i și x_i sînt liniare și de prim ordin; deci pot fi evaluate cu reduceri ciclice folosind metoda pentru recurența liniară generală de prim ordin descrisă în § 5.2.4. Nu sînt necesare discuții suplimentare. Recurența pentru w_i, deși tot o recurență cu 2 termeni, este neliniară deoarece leagă w_i de $(b_i - a_i w_{i-1})^{-1}$. Prin urmare reducerea ciclică nu poate fi aplicată direct pentru introducerea paralelismului în această recurență. Totuși, după unele transformări, aceasta se poate aplica.

Pentru w_i recurența este :

$$w_1 = c_1/b_1$$

$$w_i = c_i/(b_i - a_i w_{i-1}), \quad i=2, 3, \ldots, n-1 \qquad (5.43\ a)$$

Dacă introducem

$$w_i = y_i/y_{i+1} \qquad (5.43\ b)$$

și rearanjăm, obținem :

$$a_i y_{i-1} + b_i y_i + c_i y_{i+1} > 0, \quad i=2, 3, \ldots, n-1 \qquad (5.44\ a)$$

cu

$$y_1 = 1, \ y_2 = -c_1/b_1 \qquad (5.44\ b)$$

Ecuația (5.44 a), fără a surprinde, este forma omogenă a cuațiilor originale. Ea este o recurență liniară de ordinul 2 (sau cu 3 termeni). De aici problema

găsirii lui w_i este aceeaşi cu cea a rezolvării ecuaţiilor originale. În § 5.4.3 vom arăta cum se poate folosi direct reducerea ciclică pentru rezolvarea unor astfel de recurenţe. Algoritmul de dublare recursivă, deşi foloseşte reducerea ciclică, diferă întrucîtva, cum vom arăta în continuare.

Pentru a păstra generalitatea, considerăm recurenţa de ordinul 2:

$$a_i y_{i-1} + b_i y_i + c_i y_{i+1} = k_i \qquad (5.45\ a)$$

Ecuaţia (5.45 a) poate fi exprimată cum urmează:

$$\begin{pmatrix} y_i \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ -a_i/b_i & -c_i/b_i \end{pmatrix} \begin{pmatrix} y_{i-1} \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ k_i/c_i \end{pmatrix} \qquad (5.45\ b)$$

sau

$$v_i = Q_i v_{i-1} + h_i\ ,\ i = 2, 3, \ldots, n-1 \qquad (5.45\ c)$$

unde

$$v_i = \begin{pmatrix} y_i \\ y_{i+1} \end{pmatrix} \quad Q_i = \begin{pmatrix} 0 & 1 \\ -a_i/b_i & -c_i/b_i \end{pmatrix} \quad h_i = \begin{pmatrix} 0 \\ k_i/c_i \end{pmatrix} \qquad (5.45\ d)$$

cu

$$v_1 = \begin{pmatrix} 1 \\ -(c_1/b_1) \end{pmatrix}$$

Ecuaţia (5.45 c) este o recurenţă liniară de prim ordin pentru vectorul v_i a ecuaţiei generale (5.11 a) exceptînd că factorul multiplicant este acum o matrice. Recurenţa poate fi rezolvată folosind procedura de reducere ciclică descrisă pentru recurenţa scalară în § 5.2.4, cu o interpretare adecvată în termeni de vectori şi matrice. Aflînd v_i (luînd $h_i = 0$), valorile pentru y_i se cunosc, iar w_i se calculează cu ecuaţia (5.43 b). Astfel se încheie algoritmul cu dublare recursivă pentru evaluarea paralelă a descompunerii LU a unui sistem tridiagonal cu recurenţele de eliminare gausiană.

Se poate încerca rezolvarea ecuaţiilor originale cu această metodă, deoarece ele sînt aceleaşi cu ecuaţia (5.45 a). Dar această abordare este imposibilă deoarece lipseşte valoarea iniţială v_1. Aceasta se întîmplă deoarece un astfel de sistem tridiagonal rezultă din PDE de ordinul 2 cu condiţii la cele 2 margini (se pot considera y_0 şi y_{n+1} cunoscute). Totuşi, recurenţa poate fi evaluată progresiv dacă se dau condiţiile iniţiale pentru două valori adiacente, fie y_0 şi y_1, ce definesc o valoare de start pentru v_1. Dacă sînt disponibile astfel de valori iniţiale, metoda prezentată poate fi folosită pentru rezolvarea paralelă a recurenţei liniare generale de ordinul 2. Această situaţie intervine dacă ecuaţiile rezultă dintr-o problemă cu valori iniţiale de ordinul 2, mai degrabă decît din problema prezentată, cu valori la margini (boundary-value).

Pentru a estima numărul operaţiilor vom face supoziţia că paralelismul îşi păstrează permanent valoarea *n*. Atunci, neglijînd constantele, avem :

18 $\log_2 n$ operaţii pentru obţinerea lui v_i
3 $\log_2 n$ operaţii pentru calcularea lui g_i
3 $\log_2 n$ operaţii pentru obţinerea lui x_i
total $24 \log_2 n$ operaţii cu paralelism n (5.46a)

Timpul de execuţie al algoritmului pe un calculator definit de $n_{1/2}$ este proporţional cu

$$t_{RD} \approx 24\,(n_{1/2} + n)\log_2 n \tag{5.46b}$$

5.4.3 Reducerea ciclică

Hockney (1965) împreună cu Golub au folosit primii reducerea ciclică pentru rezolvarea ecuaţiilor tridiagonale. Metoda a fost implementată pe un calculator serial, **IBM** 7090, şi a fost preferată eliminării gausiene, deoarece reducerea ciclică consideră în mod periodic condiţiile la margine, într-un mod mult mai ordonat, eliminînd necesitatea calculării vectorilor auxiliari. S-au rezolvat ecuaţii ce rezultă din aproximarea cu diferenţe finite a ecuaţiei Poisson, cu aceiaşi coeficienţi în fiecare nod al caroiajului. Aceste ecuaţii sînt prezentate în detaliu în § 5.6, Pentru comoditate s-a luat un număr de noduri de caroiaj şi deci de ecuaţii, o putere a lui 2. Nici una din aceste restricţii nu este necesară în cadrul metodei, au arătat Swarztrauber (1974) şi Sweet (1974, 1977). Presupunem aici, pentru simplitate, că $n = n' - 1$, unde $n' = 2q$ şi q este un întreg, şi rezolvăm problema coeficienţilor generali definită de ecuaţia (5.37).

Scriind 3 ecuaţii adiacente avem pentru $i = 2, 4, \ldots, n'-2$:

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i &= k_{i-1} \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= k_i \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= k_{i+1} \end{aligned} \tag{5.47}$$

Ecuaţiile speciale de încheiere sînt corect introduse dacă se ia $x_0 = x_{n'} = 0$. Dacă prima din aceste ecuaţii este înmulţită cu $\alpha_i = -a_i/b_{i-1}$, iar ultima cu $\gamma_i = -c_i/b_{i+1}$, şi se adună cele trei ecuaţii, variabilele x_{i-1} şi x_{i+1} se elimină. Se obţine :

$$a_i^{(1)}x_{i-2} + b_i^{(1)}x_i + c_i^{(1)}x_{i+2} = k_i^{(1)} \tag{5.48 a}$$

unde

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1} \\ c_i^{(1)} &= \gamma_i c_{i+1} \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \gamma_i a_{i+1} \\ k_i^{(1)} &= k_i + \alpha_i k_{i-1} + \gamma_i k_{i+1} \end{aligned} \tag{5.48 b}$$

Ecuațiile (5.48) leagă fiecare a doua variabilă și, dacă se scriu pentru $i = 2, 4, \ldots, n'-2$, formează un sistem tridiagonal de ecuații de aceeași formă cu ecuațiile originale (5.47), dar cu coeficienți diferiți $(a^{(1)}, b^{(1)}, c^{(1)})$. În mare numărul de ecuații s-a redus la jumătate. Evident procesul poate continua recursiv, pînă ce, după $\log_2(n')-1$ etape de reducere, rămîne numai ecuația centrală pentru $i = n'/2$. Această ecuație este:

$$a^{(r)}_{n'/2}x_0 + b^{(r)}_{n'/2}x_{n'/2} + c^{(r)}_{n'/2}x_{n'} = k^{(r)}_{n'/2} \tag{5.49 a}$$

unde exponentul $r = \log_2(n') - 1$ indică nivelul de reducere. Deoarece $x_0 = x_n = 0$, soluția pentru ecuația centrală se obține prin împărțire

$$x_{n'/2} = k^{(k)}_{n'/2}/b^{(r)}_{n'/2} \tag{5.49 b}$$

Celelalte necunoscute se pot determina cu o procedură de substituire. Deoarece se cunosc x_0, $x_{n'/2}$ și $x_{n'}$, necunoscutele echidistante dintre ele pot fi aflate din ecuațiile etapei $r-1$ cu

$$x_i = (k_i^{(r-1)} - a_i^{(r-1)}x_{i-n'/4} - c_i^{(r-1)}x_{i+n'/4})/b_i^{(r-1)}.$$

pentru $i = n'/4$ și $3n'/4$. Această procedură se repetă pînă ce, în final, se află toate necunoscutele pare.

Procedura de reducere ciclică incubă calcularea recursivă a noilor coeficienți precum și a termenilor din membrul drept, pentru etapele $l = 1, 2, \ldots, q-1$, din

$$\begin{aligned} a_i^{(l)} &= \alpha_i a^{(l-1)}_{i-2^{(l-1)}} \\ c_i^{(l)} &= \gamma_i c^{(l-1)}_{i+2^{(l-1)}} \\ b_i^{(l)} &= b_i^{(l-1)} + \alpha_i c^{(l-1)}_{i-2^{(l-1)}} + \gamma_i a^{(l-1)}_{i+2^{(l-1)}} \\ k_i^{(l)} &= k_i^{(l-1)} + \alpha_i k^{(l-1)}_{i-2^{(l-1)}} + \gamma_i k^{(l-1)}_{i-2^{(l-1)}} \end{aligned} \tag{5.50}$$

unde

$$\alpha_i = -a_i^{(l-1)}/b^{(l-1)}_{i-2^{(l-1)}}$$

$$\gamma_i = -c_i^{(l-1)}/b^{(l-1)}_{i+2^{(l-1)}}$$

și

$$i = 2^l \text{ pas } 2^l \text{ pînă la } n'-2^l$$

cu valorile inițiale $a_i^{(0)} = a_i$, $b_i^{(0)} = b_i$ și $c_i^{(0)} = c_i$, urmate de substituirea recursivă a soluției pentru $l = q, q-1, \ldots, 2, 1$, de la

$$x_i = (k_i^{(l-1)} - a_i^{(l-1)}x_{i-2^{(l-1)}} - c_i^{(l-1)}x_{i+2^{(l-1)}})/b_i^{(l-1)} \tag{5.51}$$

unde

$$i = 2^{(l-1)} \text{ pas } 2^l \text{ pînă la } n'-2^{(l-1)}$$

şi $x_0 = x_{n'} = 0$ cînd intervin în ecuaţie. Diagrama de transfer pentru acest algoritm este prezentată în fig. 5.7 pentru cazul $n' = 8$. S-a definit vectorul $p_l = (a_l, b_l, c_l, k_l)$ pentru a indica valorile calculate cu ecuaţiile (5.50)

Numărul de operaţii necesare pentru evaluarea ecuaţiilor (5.50) este

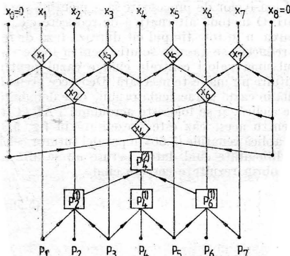


Fig. 5.7 Diagrama de transfer pentru algoritmul de reducere ciclică serială (SERICR), pentru $n' = 8$. Dreptunghiurile indică evaluarea ecuaţiei (5.50), iar romburile evaluarea ecuaţiei (5.51). Variabilele calculate sînt scrise în căsuţe cu notaţia $p_l = (a_l, b_l, c_l, k_l)$.

12 cu paralelism $n'2^{-l} - 1$ pentru $l = 1, 2, \ldots, \log_2(n') - 1$ 5.52)

Timpul de execuţie pentru un calculator definit de $n_{1/2}$ este proporţional cu

$$t_{5.50} = 12 \sum_{l=1}^{\log_2(n')-1} (n_{1/2} + n'2^{-l} - 1)$$

$$= 12[n' + n_{1/2}\log_2(n') - n_{1/2} - \log_2(n') - 1] \qquad (5.53\ a)$$

Pentru comparaţii vom păstra numai termenii de ordinul n' şi $\log_2 n'$ deci pentru $n_{1/2}$ şi $\log_2 n' > 1$ aproximativ :

$$t_{5.50} \simeq 12(n' + n_{1/2}\log_2 n') \qquad (5.53\ b)$$

Evaluarea ecuaţiei (5.51) necesită

5 operaţii cu paralelism $n2^{-l}$ pentru $l = \log_2 n', \ldots, 2, 1$

Deci

$$t_{5.51} = 5 \sum_{l=0}^{\log_2 n'} (n_{1/2} + n \cdot 2^{-l}) \qquad (5.54\ a)$$

$$= 5[n' + n_{1/2}\log_2(n') - 1] \qquad (5.54\ b)$$

$$\simeq 5(n' + n_{1/2}\log_2 n') \qquad (5.54\ c)$$

Maniera de realizare a reducerii ciclice tocmai prezentată se caracterizează prin numărul cel mai mic de operaţii aritmetice scalare, deci este cea mai bună soluţie pentru un calculator serial. De aceea vom numi acest

algoritm ca variantă serială a reducerii ciclice, și folosim acronimul SERICR. Timpul total de execuție este proporțional cu :

$$t_{SERICR} = t_{5.50} + t_{5.51} \simeq 17(n' + n_{1/2} \log_2(n')) \quad (5.55)$$

Este de dorit ca în cazul masivelor de procesoare să se mențină paralelismul la valori cît mai mari. O metodă alternativă de reducere ciclică menține paralelismul la valoarea n în tot timpul cît durează faza de reducere. La ultimul nivel de reducere se găsește soluția pentru toate variabilele în paralel — în locul unei valori centrale cum e cazul pentru SERICR — iar faza de substituire nu mai este necesară. Deoarece această metodă este foarte convenabilă în cazul paracalculatorului, este denumită varianta paralelă de reducere ciclică, și se folosește acronimul PARACR.

Diagrama de transfer pentru acest caz este prezentată în fig. 5.8. La fiecare nivel de reducere se aplică ecuațiile (5.50) în paralel tuturor celor n ecuații. Poate interveni o dificultate cînd datele cerute nu se află în domeniul $1 \leqslant i \leqslant n$. Totuși, se obțin rezultate corecte cînd

$$\left.\begin{array}{l} a_i^{(1)} = c_i^{(1)} = k_i^{(1)} = 0 \\ b_i^{(1)} = 1 \end{array}\right\} \text{ pentru } i \leqslant 0 \text{ și } i \geqslant n+1 \quad (5.56a)$$

sau

$$\mathbf{p}_i^{(1)} = (0, 1,, 0, 0)$$

Cînd se introduc în ecuațiile (5.48a) valorile de mai sus, se obține ecuația

$$x_i = 0 \text{ pentru } i \leqslant 0 \text{ și } i \geqslant n + 1 \quad (5.56b)$$

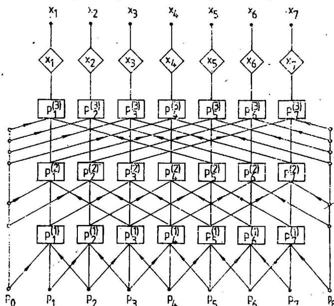


Fig. 5.8 Diagrama de transfer pentru algoritmul de reducere ciclică paralelă (PARACR) pentru $n' = 8$. Vectorul particular $p_0 = (0, 1, 0, 0)$

care furnizează valorile de margine corecte. Prin urmare putem considera fie rezolvarea sistemului inițial de ecuații, care este finit, sau, alternativ un sistem infinit extins cu coeficienții (5.56a). Se adaugă în plus ecuații ca cele notate cu (5.56b) care sînt în afara domeniului problemei inițiale. Fiecare punct de vedere este corect, dar ultimul este mai apropiat de varianta paralelă a reducerii ciclice deoarece definește valorile cerute pentru p_i și x_i în afara problemei definite la început.

După calcularea lui $p_i^{(q)}$ (sînt necesare numai valorile lui b_i și k_i), se obține din ecuația (5.51) x_i:

$$x_i = k_1^{(q)}/b_1^{(q)} \tag{5.57}$$

Termenii în x din membrul drept al ecuației (5.51) nu intervin, deoarece, la acest nivel de reducere, ei se referă la valori în afara domeniului $1 \leq i \leq n$ și conform ecuației (5.56b) sînt 0.

Numărul operațiilor necesare pentru algoritmul PARACR este evident

$$12[\log_2(n')-1] \text{ cu paralelism } n \tag{5.58a}$$

iar timpul de execuție este proporțional cu

$$t_{PARACR} = 12(n_{1/2} + n)[\log 2(n')-1]$$
$$\simeq (12(n_{1/2} + n)\log_2 n' \tag{5.58b}$$

Un avantaj al algoritmului de reducere ciclică este că, în anumite condiții, procesul de reducere poate fi oprit înainte de sfîrșit fără a se pierde din precizie. Acest lucru este posibil dacă sistemul tridiagonal este suficient de mult diagonalizat. Să definim această proprietate în cazul sistemului inițial (5.47) ca minimul în cazul tuturor ecuațiilor a rapoturilor $|b_i|/|a_i|$ și $|b_i|/|c_i|$, și o notăm cu δ. Putem apoi lua în considerare rezolvarea sistemului mai simplu de ecuații cu coeficienți constanți:

$$ax_{i-1} + bx_i + ax_{i+1} = k_i \tag{5.59}$$

cu $|b/a| = \delta$, care este cel mult egal în diagonalizare cu originalul. Dacă acest sistem de ecuații mai simplu poate fi rezolvat cu o anumită aproximație, atunci ecuațiile originale pot fi rezolvate cu mai multă precizie. Recurența de reducere ciclică (5.50) este în acest caz

$$a(0) = a$$

și apoi

$$\left.\begin{aligned} a^{(l)} &= -(a^{(l-1)})^2/b^{(l-1)} \\ b^{(l)} &= b^{(l-1)} - 2(a^{(l-1)})^2/b^{(l-1)} \end{aligned}\right\} l = 1\ 2, \ldots, \log_2(n') - 1 \qquad \begin{aligned}(5.60a)\\(5.60b)\end{aligned}$$

unde indicele i este eliminat deoarece coeficienții sînt aceiași pentru toate ecuațiile și folosim faptul că $c^{(l)} = a^{(l)}$. Împărțind ecuația (5.60b) la ecuația (5.60 a) se obține relația de recurență în cazul diagonalizării dominante:

$$\delta^{(0)} = \delta$$
$$\delta^{(l)} = |b^{(l)}|/|a^{(l)}|$$
$$= |(\delta^{(l-1)})^2 - 2|, \quad l = 1, 2, \ldots, \log_2(n')-1 \tag{5.61a}$$

De aici, dacă inițial $\delta' > 2$ ea va crește de cîte 4 ori cel puțin la fel de rapid cu ecuația (5.61a) și

$$\delta^{(l)} \approx \delta^{2^l} \text{ pentru } \delta >> 2 \tag{5.61b}$$

Aproximàrea cu diferențe finite a ecuației Helmholtz,

$$\frac{d^2\Phi}{dx^2} - \beta\Phi = s(x) \tag{5.62a}$$

pentru o distanță în caroiaj de h este

$$\Phi_{i-1} - (2 + \beta h^2)\,\Phi_i + \Phi_{i+1} = s_i h^2 \tag{5.62b}$$

Deci

$$\delta = 2 + \beta h^2 \tag{5.62c}$$

și pentru $\beta > 0$ ecuațiile diferențiale verifică condiția de creștere a diagonalizării odată cu avansarea procesului de reducere. Ecuațiile armonice care se obțin prin rezolvarea ecuațiilor cu derivate parțiale sînt un set important de ecuații ce cad în această categorie (vezi § 5.6.2).

Dacă în orice etapă a reducerii inversa măsurii diagonalizarii este sub precizia dorită (sau a calculatorului folosit), și se știe că soluția x este de ordinul unității, procesul de reducere poate fi oprit. Soluția ecuației (5.51) este

$$x_i = k_i^{(l-1)}/b_i^{(l-1)} \pm (x_{i-2^{(l-1)}} + x_{i+2^{(l-1)}})/\delta^{(l-1)} \tag{5.63a}$$

și, prin postulat termenii în x din membrul drept pot fi neglijați în raport cu membrul stîng. Soluția poate fi găsită la acest nivel prin simpla împărțire :

$$x_i = k_i^{(l-1)}/b^{(l-1)} \tag{5.63b}$$

și în PARACR se cunoaște soluția, sau în SERICR poate începe substituirea.

Pentru valori mari ale lui δ (să zicem > 3), nivelul $\hat{l}$ la care se poate opri procesul de reducere se obține folosind ecuația (5.61b) :

$$\delta^{(\hat{l})} \simeq \delta^{2^{\hat{l}}} = \varepsilon^{-1} \tag{5.64a}$$

unde ε este eroarea relativă admisă pentru soluție, sau

$$\hat{l} = \log_2[(\log \varepsilon^{-1})/(\log_2 \delta)]$$

$$= \log_2(\log_2 \varepsilon^{-1}) - \log_2(\log_2 \delta) \tag{5.64b}$$

Primul termen arată că o condiție de precizie mai mare (ε^{-1} mai mare) impune execuția mai multor nivele de reducere, iar al doilea termen că numărul nivelelor e redus odată cu creșterea măsurii de diagonalizare.

Dacă δ are valori apropiate de 2, trebuie evaluată recurența (5.61a). În orice caz, soluția practică este de a măsura diagonalizarea $\delta^{(l)}$ la fiecare nivel de reducere pe baza valorilor $a^{(l)}$, $b^{(l)}$, $c^{(l)}$ și de a opri procesul de reducere cînd este satisfăcută ecuația (5.64a).

Desigur, nu se cîștigă nimic atît timp cît nu este îndeplinită condiția $\hat{l} < \log_2(n')-1$, numărul maxim de nivele pentru reducerea completă. Se ajunge la rezultatul că o reducere trunchiată poate produce economii dacă:

$$n' > \hat{n} \tag{5.65a}$$

unde

$$\hat{n} = 2^{\hat{l}} = (\log_2 \varepsilon^{-1})/(\log_2 \delta) \quad \text{pentru } n' >> 1 \tag{5.65b}$$

Dacă considerăm exemplul $\delta = 2^{-20} \simeq 10^{-6}$ (32 biți în simplă precizie pe IBM 360) și $\delta = 4$ (că în cazul ecuațiilor armonice din § 5.62), obținem

$$\hat{l} = 3{,}32, \qquad \hat{n} = 10 \tag{5.65c}$$

Este clar că reducerea trunchiată poate produce economii avantajoase chiar pentru un număr mic de ecuații, cu condiția să fie suficient de diagonalizate. Este probabil avantajos de introdus un test pentru condiția (5.64a) în fiecare subrutină de reducere ciclică. Economiile de timp de execuție pot fi substanțiale dacă sînt implicate sute sau mii de ecuații.

5.4.4 Alegerea algoritmului

În continuare vom discuta modul cum se alege cel mai bun algoritm pentru rezolvarea a *m* sisteme tridiagonale, a cîte *n* ecuații cu *n* necunoscute fiecare. Alternativele disponibile, ca și concluziile, sînt tipice pentru modul de alegere a celui mai bun algoritm pentru rezolvarea oricărei probleme pe un calculator paralel. Vom compara din nou performanțele algoritmilor pentru un calculator cu un $n_{1/2}$ finit. Vom presupune, ca de obicei, existența unei memorii infinite fără conflict de acces. Pentru un calculator serial se consideră un singur obiectiv, minimizarea numărului de operații aritmetice. Pentru un calculator paralel există mai multe obiective, deci alegerea metodei celei mai bune devine mai complexă. În cazul problemei anterioare a *m* sisteme tridiagonale, se poate alege cel mai bun algoritm secvențial și aplica celor *m* sisteme în paralel sau să se ia cel mai bun algoritm paralel pentru rezolvarea unui singur sistem care să se aplice secvențial sau în paralel celor m sisteme. În ultimul caz apare complicația că cel mai bun algoritm depinde de caracteristicile de paralelism ale hard-ului (valoarea lui $n_{1/2}$). Prima opțiune nu este disponibilă pentru un calculator serial și toate calculatoarele au aceeași valoare pentru $n1/2$ (= 0).

Începem prin a lua în considerare cel mai bun algoritm pentru rezolvarea unui singur sistem tridiagonal de n ecuații. Folosind ecuațiile (5.46b), (5.58b) și (5.55) și ignorînd diferența neimportantă dintre n' și n se determină numărul de operații și lungimea vectorului mediu $\bar{n}$ al algoritmului în discuție ca:

(1) pentru dublare recursivă (**RD**), numerele de operaţii sînt

$$s = 24n\log_2 n \quad q = 24\log_2 n$$

De aici, folosind

$$T_{RD} \propto s + n_{1/2}\, q \quad P_{RD} = T^{-1}\ \bar{n} = s/q$$

obţinem

$$P_{RD} \propto [(24(n + n_{1/2})\log_2 n])^{-1} \quad \bar{n}_{RD} = n \tag{5.66a}$$

(2) pentru reducere ciclică cu lungimea vectorului menţinută la n şi nici o substituire (denumită **PARACR**),

$$s = 12n\log_2 n \qquad q = 12\log_2 n$$

de unde

$$P_{PARACR} \propto [12(n + n_{1/2})\log_2 n]^{-1} \quad \bar{n}_{PARACR} = n \tag{5.66b}$$

(3) pentru reducere ciclică cu lungimea vectorului înjumătăţindu-se la fiecare nivel de reducere şi cu o fază de substituire (denumit **SERICR),**

$$s = 17\,n \quad q = 17\log_2 n$$

de aici

$$P_{SERICR} \propto [17(n + n_{1/2}\log_2 n)]^{-1} \quad \bar{n}_{SERICR} = n/\log_2 n \tag{5.66c}$$

Mai sus am ignorat posibilele economii prin reducere trunchiată care ar trebui cu siguranţă luate în considerare dacă sistemele sînt puternic diagonalizate.

Algoritmul de dublare recursivă are o performanţă mai slabă decît orice variantă a reducerii ciclice, de aceea nu-l vom mai discuta. Desigur, ar putea fi folosit pentru găsirea descompunerii **LU** a ecuaţiilor. Reducerea ciclică este întotdeauna mai bună pentru problema enunţată a rezolvării ecuaţiilor. Deci, trebuie să se aleagă între variantele paralelă şi serială ale reducerii ciclice (**PARACR**, respectiv **SERICR**).

Cunoscînd numerele de operaţii *s* şi *q*, diagrama de fază algoritmică poate fi trasată folosind ecuaţia (5.5). Varianta paralelă are o performanţă mai bună sau egală cînd $P_{PARACR} \geqslant P_{SERICR}$ care conduce la relaţia :

$$n_{1/2}/n \geqslant 2{,}4\ \ (1-1{,}42/\log_2 n) \tag{5.67a}$$

Egalitatea produce linia de performanţă egală. Ea apare în fig. 5.9 împreună cu domeniile unde fiecare algoritm este superior. Pentru $n_{1/2}$ mari avem :

$$n_{1/2}/n \geqslant 2{,}4 \tag{5.67b}$$

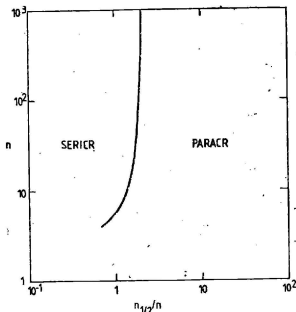


Fig. 5.9 Alegerea celui mai bun algoritm pentru rezolvarea unui singur sistem tridiagonal de n ecuații pe un calculator caracterizat de un $n_{1/2}$: SERICR, reducerea ciclică fără reducerea lungimii vectorului (Conform Hockney (1982)).

Rezultă că pentru paracalculator ($n_{1/2} = \infty$), varianta paralelă **PARACR** este cel mai bun algoritm pentru orice ordin n al ecuațiilor (de unde și numele variantei).

Pentru valori finite ale lui $n_{1/2}$ există întotdeauna un n ($\approx 0{,}42 n_{1/2}$) astfel încît pentru valori mai mari decît n varianta serială este superioară. Acest rezultat este analog celui găsit în §5.2.3 la rezolvarea recurențelor. Valoarea lui $n_{1/2}$ este o măsură a paralelismului hardware. Dacă lungimea vectorului este mult mai mare decît aceasta, pentru o problemă anume, atunci pentru acea problemă calculatorul se va comporta ca un calculator serial, adică paralelismul calculatorului este prea mic pentru a influența performanța. În această circumstanță va fi relevantă performanța pe un calculator serial: cel mai bun algoritm este cel cu numărul cel mai mic de operații aritmetice, prin urmare varianta serială a reducerii ciclice (**SERICR**).

Dacă alternativ, avem de rezolvat *m* sisteme tridiagonale a *n* ecuații fiecare, avem de ales între aplicarea fie a lui **SERICR**, fie a lui **PARACR** în paralel, respectiv serial, celor *m* sisteme, sau de a folosi cea mai bună metodă serială (recurența de eliminare gausiană descrisă în ultimul paragraf din § 5.4.1) tuturor sistemelor în paralel. Pentru calculatoare cu un paralelism natural mare $\geqslant$ m.n, cum sînt masivele de procesoare mari (**ICL DAP**), sau calculatoarele pipeline cu $n_{1/2}$ mare (**CYBER** 205), este probabil că cel mai bun algoritm este cel mai paralel. În aceste situații se compară eliminarea gausiană aplicată în paralel tuturor sistemelor (**MULTGE**) cu paralelism *m*, cu $\mathbf{SERICR}_{par}$ și $\mathbf{PARACR}_{par}$, unde se aplică algoritmul de reducere ciclică în paralel tuturor sistemelor. Pentru aceste alternative, performanța și paralismul mediu sînt:

(1) Pentru **MULTGE** s = 8nm, q = 8n, de aici

$$P_{MULTGE} \propto [8\, n(m + n_{1/2})]^{-1} \quad n_{MULTGE} = m \tag{5.68 a}$$

(2) Pentru $\mathbf{PARACR}_{par}$ s = 12 mnlog$_2$n, q = 12log$_2$n, deci

$$P_{PARACR_{par}} \propto [12(\mathrm{m}\cdot n + n_{1/2}) \log n)^{-1} \quad n_{PARACR_{par}} = m\cdot n \tag{5.68 b}$$

(3) Pentru **SERICR**$_{par}$ $s = 17\ mn$, $q = 17 \log_2 n$, de aici,

$$P_{SERICR_{par}} \propto [17\,(m \cdot n + n_{1/2}) \log_2 n)]^{-1} \quad \bar{n}_{SERICR_{par}} = m \cdot n / \log_2 n \qquad \textbf{(5.68 c)}$$

Egalînd expresiile de mai sus în perechi obținem ecuațiile ce definesc liniile de egale performanță în planul $(n_{1/2}/m, n)$. În plus, inegalitățile de mai jos definesc regiunile planului unde fiecare algoritm este mai avantajos:

$$P_{PARACR_{par}} \geqslant P_{MULTGE} \text{ cînd}$$

$$n_{1/2}/m \geqslant (1{,}5 \log_2 n - 1)/[1 - (1{,}5 \log_2 n/n)] \qquad \textbf{(5.69 a)}$$

$$\geqslant 1{,}5 \log_2 n \text{ cînd } n \to \infty$$

$$P_{SERICR_{par}} \geqslant P_{MULTGE} \text{ cînd}$$

$$n_{1/2}/m \geqslant 1{,}1/[1 - (2{,}1 \log_2 n/n)] \qquad \textbf{(5.69 b)}$$

$$\geqslant 1{,}1 \qquad \text{cînd } n \to \infty$$

$$P_{PARACR_{par}} \geqslant P_{SERICR_{par}} \text{ cînd}$$

$$n_{1/2}/m \geqslant 2{,}4\ [1 - (1{,}42/\log_2 n)] \geqslant 2{,}4 \text{ cînd } n \to \infty \qquad \textbf{(5.69 c)}$$

Liniile definite de ecuațiile (5.69) sînt prezentate în fig. 5.10, unde împart planul $(n_{1/2}/m), n)$ în regiuni în care fiecare din cele trei metode are perfor-

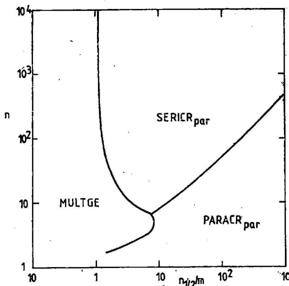


Fig. 5.10 Selecția celui mai bun algoritm pentru rezolvarea celor m sisteme tridiagonale a n ecuații pentru calculatoare cu un paralelism natural mare $\geqslant mn$. Comparăm eliminarea gausiană (MULTGE), și versiunile seriale ($SERICR_{par}$) și paralelă ($PARACR_{par}$) ale reducerii ciclice cînd se aplică în paralel tuturor celor m sisteme (Conform Hockney (1982)).

manța cea mai bună. Se observă pe diagramă că există un punct de coordonate $n \approx 7$, $n_{1/2}/m \approx 7{,}7$, un „punct triplu" unde cei trei algoritmi au aceeași performanță. Găsim că pentru $n_{1/2} = \infty$ (paracalculatorul), $\mathbf{PARACR_{par}}$ este, așa cum ne așteptam, cel mai bun algoritm pentru toți n. Oricum, pentru un $n_{1/2}$ finit și $n_{1/2} > 10$ m există întotdeauna un n, de la care $\mathbf{SERICR_{par}}$ este mai avantajos. Dacă numărul m al sistemelor este mai mare decît $n_{1/2}$ se găsește că este întotdeauna cel mai bine să se aplice algoritmul serial în paralel celor n sisteme. În regiunea $1 < n_{1/2}/m < 10$ toți cei trei algoritmi pot fi mai avantajoși conform diagramei prezentate.

Pentru calculatoare cu un paralelism limitat la aproximativ n sau m, există insuficient paralelism pentru aplicarea algoritmilor **SERICR** sau **PARACR** în paralel. Această situație caracterizează calculatoare ca de exemplu **CRAY X-MP** (care are un paralelism natural egal cu 64) cînd sînt folosite pentru rezolvarea a 64 sisteme tridiagonale cu lungimea 64. În acest caz trebuie să comparăm $\mathbf{SERICR_{seq}}$ (cînd algoritmul respectiv de reducere ciclică se aplică secvențial celor m sisteme) și $\mathbf{PARACR_{seq}}$ cu cea mai bună metodă serială aplicată în paralel (**MULTGE**). În aceste cazuri performanța și lungimea vectorilor medii sînt:

$$P_{PARACR_{seq}} \propto [12\,m(n+n_{1/2})\log_2 n]^{-1} \quad \bar{n}_{PARACR_{seq}} = n \qquad (5.70\ a)$$

$$P_{SERICR_{seq}} \propto [17\,m(n+n_{1/2}\log_2 n)]^{-1} \quad \bar{n}_{SERICR_{seq}} = n/\log_2 n \qquad (5.70\ b)$$

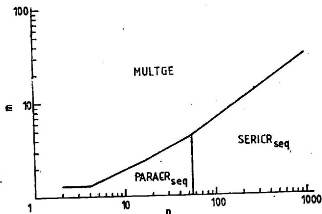


Fig. 5.11 Selecția celui mai bun algoritm pentru rezolvarea a m sisteme tridiagonale a n ecuații pentru un calculator cu paralelismul limitat la aproximativ m sau n (aici $n_{1/2} = 100$). Comparăm eliminarea gausiană aplicată în paralele tuturor celor m sisteme (MULTGE) cu versiunile serială ($SERICR_{seq}$) și paralel ($PARACR_{seq}$) ale reducerii ciclice aplicate secvențial celor m sisteme.

Fig. 5.11 ilustrează comparația celor doi algoritmi cu **MULTGE** pentru cazul $n_{1/2} = 100$. Linia verticală ce separă $\mathbf{PARACR_{seq}}$, și $\mathbf{SERICR_{seq}}$ se obține din egalitatea corespunzătoare ecuației (5.67 a) sau din fig. 5.9 și este valabilă pentru toți *m*. Eliminarea gausiană multiplă

va prezenta o performanță mai bună sau egală cu cea a variantei paralele a reducerii ciclice aplicată secvețial cînd

$$P_{MULTGE} \geqslant P_{PARACR_{seq}}$$

sau cînd

$$m \geqslant n_{1/2}[(1{,}5 \log_2(n) - 1) + (1{,}5\, n_{1/2} \log_2 n)/n]^{-1}$$

$$\geqslant 0{,}67\; n/\log_2 n \qquad \text{cînd } n_{1/2} \to \infty \qquad (5.70\ c)$$

În mod similar **MULTGE** este superior lui SERICR seq cînd

$$P_{MULTGE} \geqslant P_{SERICR_{seq}}$$

sau cînd

$$m \geqslant n_{1/2}[1{,}125 + 2{,}125\; (n_{1/2} \log_2 n)/n]^{-1} \qquad (5.70\ d)$$

Se obține, în sens larg, că aplicarea multiplă a algoritmului gausian secvențial este cea mai bună cînd numărul sistemelor depășește 1/10 din numărul ecuațiilor fiecărui sistem. Această situație se întîlnește în cele mai multe aplicații ce rezultă la rezolvarea ecuațiilor cu derivate parțiale (vezi § 5.6). O mică zonă de pe diagramă favorizează varianta paralelă a reducerii ciclice, care se și micșorează odată cu scăderea lui $n_{1/2}$. Într-adevăr pentru $n_{1/2} \leqslant 10$, **PARACR**$_{seq}$ nu este niciodată cel mai bun algoritm.

5.5 Transformate

Transformatele matematice joacă un rol important în analiza matematică și numerică. Printre acestea, cele mai importante, sînt transformata Fourier finită (vezi Bracewell 1965) și transformatele teoretice numerice înrudite, deoarece pentru evaluarea lor există algoritmi rapizi. Algoritmul transformatei Fourier rapide (FFT) (vezi § 5.5.1) obține toate componentele transformării a n valori în 0 (nlog₂n) operații aritmetice scalare, în comparație cu $0(n^2)$ operații pentru alte transformate. Transformata teoretică numerică (NTT) (vezi § 5.5.6) are nu numai un număr logaritmic de operații, dar și avantajul folosirii adunărilor întregi și deplasărilor; nu se folosesc operații de înmulțire. De aceea, NTT este foarte avantajoasă pentru arhitecturile orientate pe bit, ca de exemplu, **ICL DAP** sau Goodyear **STARAN** (vezi capitolul 3).

Iată unele aplicații ale transformatei Fourier:

(a) Transformata Fourier ca o funcție de timp (analiza seriilor de timp pentru a determina spectrul frecvențelor. De obicei, transformata Fourier este cea mai bună soluție pentru calcularea autocorelației și intercorelației unor serii de timp, în timp ce filtrarea datelor se realizează cel mai bine în domeniul frecvențelor (de exemplu reducerea sau eliminarea zgomotelor de frecvență înaltă). Pentru o tratare completă recomandăm Blcackman și Tukey (1959).

b) Transformata Fourier ca o funcţie de spaţiu pentru a obţine spectrul numerelor de undă. În multe situaţii o analiză a spectrului undelor conduce la cea mai bună înţelegere a unui fenomen (aflarea unor lungimi de undă instabile, de exemplu). Orice sistem liniar (sau perturbaţii de mică amploare ale unui sistem neliniar) poate fi complet descris cu ajutorul ecuaţiei de dispersie, cu alte cuvinte frecvenţa de oscilaţie ca o funcţie de lungimea de undă a perturbaţiei. Evaluarea acestei relaţii impune o analiză Fourier atît în domeniul timpului cît şi al spaţiului, pentru un mare număr de lungimi de undă şi frecvenţe.

c) Corelaţiile spaţiale între particulele, să spunem, ale unui lichid sînt descrise cu perechea funcţiilor de corelaţie. Cu alte cuvinte, probabilitatea ca două particule să se afle la o anumită distanţă ca o funcţie de acea distanţă. Razele x şi difracţia neutronilor permit determinarea factorului de structură, care este transformata Fourier a corelaţiei spaţiale. Pentru a interpreta experimentele sînt necesare multe transformări ale lungimii de undă în spaţiul coordonatelor.

(d) Simularea particulelor prin metode de dinamică moleculară (vezi Hockney şi Eeastwood 1988, capitolul 12), produce orbitele unui mare număr de atomi dintr-un lichid. Interpretarea proprietăţilor dinamice ale unui lichid şi comparaţia cu teoria se realizează cel mai bine folosind factorul de structură dinamică. Acesta se determină prin analiza Fourier a coordonatelor particulei în spaţiu şi timp.

e) Rezolvarea ecuaţiilor liniare cu derivate parţiale intervine frecvent în spaţiul lungimilor de undă (aşa numitele metode, Galerckin sau spectrale, vezi Orszag 1971). Termenii neliniari sînt incluşi prin transformare în spaţiul coordonatelor, oglindind interacţia neliniară, iar apoi transformaţi înapoi în spaţiul lungimilor de undă. Evident, astfel de metode pot fi folosite dacă există şi se pot folosi metode rapide de evaluare a transformatelor.

f) C a un caz particular al punctului e) se găseşte că unele din cele mai rapide metode pentru rezolvarea formei discrete a ecuaţiei Poisson în geometriile simple se bazează pe o transformată Fourier parţială ce foloseşte FFT (algoritmul **FACR** propus de Hockney (1965) prezentat în § 5.6.2). În aceste circumstanţe se impune execuţia mai multor transformate Fourier în paralel, de exemplu pentru transformarea datelor corespunzătoare fiecărei linii din cele 64 ale unui caroiaj 64×64, în paralel. În cazul unei aplicaţii meteorologice cînd se foloseşte o procedură de eşantionare în timp, sînt necesare mii de transformate Fourier la fiecare moment de eşantionare (Temperton 1979 b).

g) Rezolvarea problemelor cu mai multe cîmpuri poate fi exprimată ca o convoluţie între o distribuţie sursă şi o funcţie-influenţă care descrie influenţa unui punct sursă asupra soluţiei. Folosind teorema convoluţiei, se obţine că transformata Fourier a soluţiei este produsul, lungime de undă cu lungime] de undă, între transformata Fourier a sursei şi cea a funcţiei-influenţă. Soluţia poate fi găsită prin : i) considerarea transformatei sursei; ii) înmulţirea ei cu transformata funcţiei-influenţă care poate fi pre-calculată şi memorată ; şi, în sfîrşit, iii) calcularea transformatei inverse pentru a obţine soluţia. O astfel de metodă este convenabilă numai prin folosirea FFT, şi a fost folosită intens pentru calculul potenţialului gravitaţional în cazul unui sistem izolat de stele, adică o galaxie (Hockney 1970, Eastwood şi Brownrigg 1975, Brownrigg 1975), ca şi în

calculul interacţiunilor electrostatice în cristalele ionice, cînd s-a folosit algoritmul P^3M (Eastwood 1976, Amini şi Hockney 1979, Hockney şi Eastwood 1981).

Lista de aplicaţii nu este în orice caz exhaustivă. Alte aplicaţii ce folosesc filtrarea numerică, calculul invariantelor şi a densităţilor spectrale, medierea şi filtrarea, ca şi transformata Laplace sînt prezentate de Cooley et al (1967).

5.5.1. Transformata Fourier rapidă

Transformata Fourier rapidă este un termen folosit pentru un grup de metode folosite în evaluarea transformatei Fourier finite (sau analiza Fourier):

$$\bar{f}^k = \frac{1}{n} \sum_{j=0}^{n-1} \exp(-2\pi\, ijk/n)\, f_j, \quad 0 \leqslant k \leqslant n-1 \qquad (5.71\ a)$$

unde f_j sînt cei n termeni complecşi ce urmează a fi transformaţi, iar $\bar{f}^k$ cele n amplitudini armonice complexe ce rezultă. O evaluare directă a definiţiei (5.71 a) ar necesita n înmulţiri complexe şi n sume complexe pe armonică, sau un total de $8n^2$ operaţii aritmetice reale pentru evaluarea celor n armonice. Valoarea algoritmului FFT constă în reducerea numărului de operaţii la aproximativ $5n\log_2 n$ operaţii aritmetice reale. Raportul dintre performanţa algoritmului FFT şi evaluarea directă pe un calculator serial este:

$$P_{FFT}/P_{DE} = 8\,n^2/(5n \log_2 n) \approx n/\log_2 n \qquad (5.71\ b)$$

Acest raport variază de la aproximativ 18 ($n = 128$) la 102 ($n = 1024$) şi aproximativ 5×10^4 ($n \approx 10^6$). De aceea nu surprinde faptul că publicarea algoritmului FFT de către Cooley şi Tukey (1965) a condus la o revoluţie majoră în cadrul metodelor numerice. Deşi Cooley, Lewis şi Welch (1967) citează lucrări anterioare conţinînd ideea pentru metode cu $n\log_2 n$ operaţii (de exemplu Runge şi Konig 1924, Stumpff 1935, Danielson şi Lanczos 1942, Thomas 1963) acestea nu erau cunoscute pe scară largă. Înainte de 1965, transformata Fourier era considerată un proces costisitor n^2 ce putea fi folosit rareori şi de obicei evitat; după 1965 a devenit un proces relativ ieftin, mai rapid decît era înainte cu mai multe ordine de mărime.

Transformata (5.71 a) are inversa (sau sinteza Fourier):

$$f_j = \sum_{k=0}^{n-1} \exp(2\pi ijk/n)\, \bar{f}^k, \quad 0 \leqslant j \leqslant n-1 \qquad (5.72\ a)$$

ce poate fi demonstrată cu ajutorul relaţiei de ortogonalitate:

$$\sum_{k=0}^{n-1} \exp\,[2\pi\, ik\,(j-j')/n] = n\, \delta_{jj} \qquad (5.72\ b)$$

unde δ_{jj}, este simbolul Kronecker:

$$\delta_{jj} = \begin{cases} 0 & j \neq j, \\ 1 & j = j, \end{cases} \qquad (5.72\ c)$$

Ignorînd în ecuaţia (5.71 a) factorul în n^{-1}, ce poate fi inclus în afara rutinei pentru transformata Fourier, atît transformata directă cît şi cea inversă pot fi exprimate cu :

$$y_k = \sum_{j=0}^{n-1} \omega_n^{jk} x_j, \quad 0 \leqslant j \leqslant n-1 \tag{5.73 a}$$

unde

$$\omega_n = \begin{cases} \exp(-2\pi i/n) \text{ în analiză} & (5.73 \text{ b}) \\ \exp(+2\pi i/n) \text{ în sinteză} & (5.73 \text{ c}) \end{cases}$$

Asocierea semnului „—" în analiza Fourier în ecuaţia (5.73 b) ca şi a semnului „+" cu sinteza Fourier este arbitrară. Am respectat convenţiile teoriei transmisiei şi din lucrarea Bracewell (1965). Evident, alegerea este neesenţială şi poate fi ajustată prin alegerea corespunzătoare a lui ω_n. Algoritmul este acelaşi în ambele cazuri. Deoarece ω_n este a n—a rădăcină a unităţii, are proprietatea importantă că

$$\omega_n^n = 1 \tag{5.74 a}$$

şi astfel

$$\omega_n^{ns+t} = (\omega_n^n)^s \, \omega_n^t = \omega_n^t$$

unde s şi t sînt întregi. Definiţiile (5.73 b, c) arată de asemenea că orice factor comun poate fi înlocuit sau introdus atît în indicele cît şi exponentul lui ω_n^m fără a-i modifica valoarea. Astfel, de exemplu

$$\omega_n^{n/2} = \omega_1^{1/2} = \omega_2 = -1 \tag{5.74 b}$$

Funcţia ce urmează a fi transformată este frecvent reală, să o notăm cu g_j, iar transformatele sînt exprimate în termeni de sin şi cosin. Dacă presupunem că n este par :

$$g_j = \frac{1}{2} \left[a_0 + a_{n/2} (-1)^j\right] +$$

$$+ \sum_{k=1}^{n/2-1} \left[a_k \cos(2\pi \, jk/n) + bk \sin(2\pi \, jk/n)\right] \tag{5.75 a}$$

unde

$$a_k = \frac{2}{n} \sum_{j=0}^{n-1} g_j \cos(2\pi \, jk/n) \tag{5.75 b}$$

şi

$$b_k = \frac{2}{n} \sum_{j=0}^{n-1} g_j \sin(2\pi \, jk/n) \tag{5.75 c}$$

Coeficienţii transformatei reale a_k şi b_k sînt în următoarele relaţii cu părţile reale ale transformatei complexe (5.71 a) a lui g :

$$a_k = 2\ \mathrm{Re}\ (\bar{g}^k),\ b_k = -2\ \mathrm{Im}\ (\bar{g}^k) \qquad (5.75\ d)$$

Deci, o transformată reală poate fi obţinută prin execuţia unei transformate complexe de lungime n asupra unor date cu partea reală g_j şi partea imaginară zero. Părţile reale şi imaginare ale primelor n/2 armonice ale transformatei complexe produc coeficienţii transformatei reale cu ecuaţia (5.75 d). Următoarele n/2 armonice sînt redundante, fiind conjugatele complexe ale primelor n/2 armonice :

$$\bar{g}^{n-k} = (\bar{\ }^{k})^* = \mathrm{Re}(\bar{g}^k) - i\ \mathrm{Im}\ (\bar{g}^k),\ 0 \leqslant k \leqslant n/2 \qquad (5.76)$$

Metoda de mai sus de calcul al transformatei reale este evident neeconomică deoarece jumătate din datele de intrare sînt zero (toate părţile imaginare) iar jumătate din rezultate nu conţin noi informaţii şi pot fi eliminate. Numărul operaţiilor aritmetice reale scalare este $5n\log_2 n$.

O metodă mai economică de realizare a transformatei reale constă în interpretarea valorilor pare ale datelor reale de intrare ca părţile reale ale funcţiei ce urmează să fie transformată, iar valorile impare ale datelor reale de intrare ca părţi imaginare ale aceleiaşi funcţii, deci

$$f_j = g_{2j} + i\ g_{2j+1},\ 0 \leqslant j \leqslant n/2-1 \qquad (5.77)$$

Acum transformata complexă a lui f_j este de lungime n/2 şi consumă $2\times(1/2)n\log_2 n$, operaţii reale scalare. În continuare se determină coeficienţii transformatei reale astfel :
a) Se calculează pentru $k = 1, 2, \ldots, n/4-1$ transformatele celor n/2 valori pare şi n/2 valori impare :

$$\overline{\ }_{\text{pare}}k = \frac{2}{n}\sum_{j=0}^{n/2-1} g_{2j}\ \omega_{n/2}^{jk} = \frac{1}{2}\left[\bar{f}^k + (\bar{f}^{n/2-k})^*\right] \qquad (5.78\ a)$$

$$\overline{\ }_{\text{impare}}k = \frac{2}{n}\sum_{j=0}^{n/2-1} g_{2j+1}\ \omega_{n/2}^{jk} = \frac{1}{2i}\left[\bar{f}^k - (\bar{f}^{k/2-k})^*\right] \qquad (5.78\ b)$$

unde asteriscul notează conjugatul complex.
b) Se calculează pentru $k = 1, 2, \ldots, n/4-1$ transformata intermediară C_k de lungime n/2 definită cu

$$C_k = \overline{\ }_{\text{pare}}k + \omega_n^k\ \overline{\ }_{\text{impare}}k \qquad (5.79\ a)$$

$$C_{n/2-k} = \overline{\ }_{\text{pare}}k - \omega_n^k\ \overline{\ }_{\text{impare}}k \qquad (5.79\ b)$$

$$C_{n/4} = (\bar{f}^{n/4})^* \qquad (5.79\ c)$$

c) În sfîrşit se obţin coeficienţii transformatei reale din

$$a_0 = \mathrm{Re}\,(\bar{f}^0) + \mathrm{Im}\,(f^0) \qquad (5.80\ a)$$

$$a_{n/2} = \mathrm{Re}\,(\bar{f}^0) - \mathrm{Im}\,(\bar{f}^0) \qquad (5.80\ b)$$

$$\left.\begin{array}{l} a_k = \mathrm{Re}(C_k) \\ b_k = -\mathrm{Im}\,(C_k) \end{array}\right\} k = 1, \ldots, n/2-1 \qquad (5.80\ c)$$

Numărul operaţiilor aritmetice scalare reale este $2(1/2)n\log_2 n + 3(1/2)n$, unde al doilea termen apare datorită prelucrării lui $\bar{f}_k$ în ecuaţiile (5.78)—(5.80).

Calculul seriilor Fourier (5.75 a) pe baza coeficienţilor (5.75 b, c) poate fi realizat prin inversarea procedurii de mai sus astfel:
a) Facem

$$C_0 = a_0,\ C_{n/2} = a_{n/2} \qquad (5.81\ a)$$

şi

$$C_k = a_k - i\,b_k,\ k = 1, 2, \ldots, n/2-1 \qquad (5.81\ b)$$

b) Evaluăm

$$\left.\begin{array}{ll} \underline{\quad}_{\text{pare}}k = \frac{1}{2}\,(C_k + C^*_{n/2-k}) & (5.82\ a) \\ \underline{\quad}_{\text{impare}}k = \frac{1}{2}\,(C_k - C^*_{n/2-k})\,\omega_n^{-k} & (5.82\ b) \end{array}\right\} k = 1,4\ldots, n/4-1$$

c)

$$\left.\begin{array}{ll} \bar{f}^{(k)} = \underline{\quad}_{\text{pare}}k + i\,\underline{\quad}_{\text{impare}}k & (5.83\ a) \\ \bar{f}^{n/2-k} = (\underline{\quad}_{\text{pare}}k - i\,\underline{\quad}_{\text{impare}}k)^* & (5.83\ b) \end{array}\right\} k = 1, 2, \ldots, n/4-1$$

şi de asemenea

$$\bar{f}^0 = \frac{1}{2}\,(C_0 + C_{n/2}) + \frac{1}{2}\,i(C_0 + C_{n/2}) \qquad (5.83\ c)$$

$$\bar{f}^{n/4} = C^*_{n/4} \qquad (5.83\ d)$$

Apoi, se calculează sinteza Fourier complexă a celor n/2 valori complexe $\bar{f}_k$, iar cele n/2 valori complexe rezultate conţin valorile sintetizate cerute sub forma unor succesiuni de părţi reale şi imaginare.

$$f_j = \sum_{k=0}^{n/2-1} \exp\left(\frac{2\pi\, ijk}{n/2}\right) \bar{f}^k = g_{2j} + ig_{2j+1},\quad j = 0, 1, \ldots, n/2-1 \qquad (5.83\ e)$$

Procedura de mai sus implică o pre-distribuire a celor n date reale sub forma celor n/2 valori complexe, ce sînt apoi transformate cu FFT complexă. Ca şi în cazul analizei, numărul operaţiilor aritmetice scalare reale este $2 \cdot (1/2)n \log_2 n + 3 \times (1/2)n$.

Calculul transformatei Fourier reale este analizat de Cooley et al (1967) şi de Bergland (1968). O soluţie diferită, ce foloseşte varianta cu sin şi cos este descrisă de Hockney (1970). Această ultimă metodă este o amalgamare a metodelor de calcul prezentate de Runge (1903, 1905), şi Whittaker şi Robinson (1944) şi furnizează în plus faţă de transformata periodică transformatele în sin şi cos. Lucrarea lui Cooley et al (1970) abordează acest subiect împreună cu transformata Laplace. Alte lucrări privitoare la FFT sînt cele publicate de Gentleman şi Sande (1966), Singleton (1967, 1969), Uhrich (1969), ca şi Brigham (1974). În acest capitol dorim să prezentăm numai principalele caracteristici ale algoritmului FFT ca şi unele consideraţii ce afectează implementarea sa pe calculatoare paralele. Pentru atingerea acestui scop vom limita discuţia la cazul binar (sau baza 2) pentru $n = 2^q$ (unde q este un întreg), deşi algoritmul poate fi aplicat eficient oricărui n care este un produs de numere prime mici ca valoare, care, preferabil, se repetă de mai multe ori. Astfel de algoritmi sînt prezentaţi de Singleton (1969) şi Temperton (1977), (1983 a, 1983 c). O variantă a algoritmului binar, foarte potrivită prelucrării paralele a fost propusă de Pease (1968), iar performanţa multor algoritmi din cei propuşi a fost analizată comparativ de Temperton (1979 b) prin implementarea pe **CRAY-1** şi de Temperton (1984) şi Kascic (1984 b) pe **CYBER 205** Vectorizarea transformatei Fourier este tratată de Korn şi Lambriotte (1979), Wang (1980) şi Swarztrauber (1982, 1984). Jesshope (1980 a) abordează implementarea algoritmilor în baza 2 rapizi pe masive de procesoare.

Algoritmii de mai sus, variante şi extensii ale algoritmului publicat de Cooley şi Tukey (1965), pot fi numiţi *transformata Fourier rapidă convenţională*. Eficienţa lor constă în factorii de n ce se repetă de multe ori. În mod curios, mai există o formă a transformatei Fourier rapide, denumită *algoritmul cu factor prim* (prime factor algorithm — PFA), în care n este divizat în factori primi între ei irepetabili. Această metodă a fost propusă iniţial de Good (1958, 1971) şi Thomas (1963) şi dezvoltată ulterior de Kolba şi Parks (1977), Winograd (1978) şi Johnson şi Burrus (1983). Metoda este aplicată în prelucrarea semnalelor (Burrus 1977, Burrus şi Eschenbacher 1981) şi este descrisă complet în cărţile publicate de McClellan şi Rader (1779) şi Nussbaumer (1982). Temperton (1983 b, 1985, 1988) tratează implementarea PFA pe calculatoarele vectoriale curente.

Un aspect al acestor metode este manipularea formulelor algebrice pentru FFT în sensul minimizării numărului de înmulţiri. Teoria privind această tehnică a fost dezvoltată de Winograd (1978, 1980), iar metoda a fost folosită extensiv pentru calculul FFT şi al convoluţiilor de către Cooley (1982) şi Auslander şi Cooley (1986). Auslander (et al) în 1984 analizează stabilitatea numerică a algoritmilor rezultaţi.

5.5.2 Deducerea FFR

Cheia pentru deducerea algoritmului transformatei Fourier rapide este definiţia corespunzătoare a transformatelor parţiale intermediare. Vom adopta definiţia folosită de Roberts (1977), cu o modificare a notaţiei. Transformatele parţiale a n date, $f_0, f_1, \ldots, f_{n-1}$ la nivelul l al transformatei se definesc cu :

$$\overset{(l)}{\bar{f}}{}^{k}_{i} = \sum_{j=0}^{2^l-1} \omega_{2^l}^{jk} \bar{f}_{jn2^{-l}+i} \qquad (5.84\ a)$$

unde

$$j,k = 0,1, \ldots, 2^l - 1 \qquad (5.84\ b)$$

şi i este un indice întreg cu valorile

$$i = 0, 1, \ldots, n2^{-l} - 1 \qquad (5.84\ c)$$

Am definit $n2^{-l}$ transformate. Fiecare se distinge prin numărul de identificare i, un index faţă de prima dată de la care începe calculul transformatei. Datele rămase se separă de prima prin intervalul $n2^{-l}$. Lungimea fiecărei transformate este 2^l. Astfel, $\overset{(l)}{\bar{f}}{}^{k}_{i}$ este armonica a k—a a transformatei i la nivelul l (vom folosi notaţia $\overset{(l)}{\bar{f}}{}^{k}_{i}$ numai în text). Cînd $l = 0$ avem $j = k = 0$ şi

$$\overset{(0)}{\bar{f}}{}^{0}_{i} = f_i, \quad i = 0,1, \ldots, n-1 \qquad (5.85\ a)$$

De aici, transformatele de la nivelul 0 sînt datele iniţiale. La nivelul $l = q$, unde $q = \log_2 n$ avem $i = 0$ şi

$$\overset{(q)}{\bar{f}}{}^{k}_{0} = \sum_{j}^{n-1} \omega_n^{jk} f_j = n\bar{f}^k \qquad (5.85\ b)$$

Astfel, la nivelul $\log_2 n$ există numai o transformată parţială care este proporţională cu transformata completă cerută corespunzătoare tuturor datelor iniţiale n.

Este folositor să interpretăm transformatele parţiale $\overset{(l)}{\bar{f}}{}^{k}_{i}$ ca elemente ale unei matrici bidimensionale cu 2^l linii şi $n2^{-l}$ coloane, formată prin scrierea armonicelor fiecărei transformate parţiale ca un vector coloană. Apoi, se începe la nivelul 0 cu o singură linie de date şi se termină, la nivelul $\log_2 n$ cu o singură coloană de rezultate. La trecerea de la un nivel la celălalt numărul coloanelor se înjumătăţeşte, iar numărul liniilor se dublează. Nu trebuie considerat necesar ca transformatele parţiale să se memoreze în calculator ca variabilă masiv bi-dimensional. În **FORTRAN**

se folosește invariabil indexarea după o dimensiune și vom vedea că se poate economisi spațiu de memorie dacă se calculează amplitudinile armonice într-o ordine diferită de cea naturală. Se poate folosi indexarea bi-dimensională dacă limbajul posedă construcțiile paralele necesare, implementate eficient, așa cum s-a discutat în § 4.3.1 (iii).

Aritmetica algoritmului se obține prin derivarea unei recurențe pentru transformatele parțiale la nivelul $l + 1$ în termenii nivelului l:

$$\overset{(l+1)}{\overline{f_i^k}} = \sum_{j=0}^{2^{l+1}-1} \omega_{2^{l+1}}^{jk} f_{jn2^{-(l+1)}+i} \tag{5.86 a}$$

Împărțind termenii sumei în două prin notația $j = 2s + t$, cu $s = 0, 1, \ldots, 2^l$, iar $t = 0, 1$ se obține:

$$\overset{(l+1)}{\overline{f_i^k}} = \sum_{s=0}^{2^l-1} \sum_{t=0}^{1} \omega_{2^{l+1}}^{(2s+t)k} f_{(2s+t)\,n2^{-(l+1)}+i} \tag{5.86 b}$$

Reamintind definirea lui ω ca a 2^{l+1} rădăcină a unității, obținem

$$\omega_{2^{l+1}}^{(2s+t)k} = \omega_{2^{l+1}}^{tk} \cdot \omega_{2^{l+1}}^{2sk} = \omega_{2^{l+1}}^{tk}\, \omega_{2^l}^{sk} \tag{5.86 c}$$

unde, la ultimul pas, am înlocuit factorul comun 2 din indicele și exponentul celui de-al doilea factor. Înlocuind ecuația (5.86 c) în (5.86 b) și separînd termenii corespunzători lui $t = 0$ și $t = 1$ se obține:

$$\overset{(l+1)}{\overline{f_i^k}} = \sum_{s=0}^{2^l-1} \omega_{2^l}^{sk} f_{sn2^{-l}\pm i} + \omega_{2^{(l+1)}}^{k} \sum_{s=0}^{2^l-1} \omega_{2^l}^{sk} f_{sn2^{-l}+i+n2^{-(l+1)}}$$

Deci

$$\overset{(l+1)}{\overline{f_i^k}} = \overset{(l)}{\overline{f_i^k}} + \omega_{2^{l+1}}^{k} \overset{(l)}{\overline{f}}_{i+n2^{-(i+1)}} \tag{5.87 a}$$

Înlocuim pe k cu $k + 2^l$

$$\overset{(l+1)}{\overline{f_i^{k+2^l}}} = \overset{(l)}{\overline{f_i^k}} - \omega_{2^{l+1}}^{k} \overset{(l)}{\overline{f}}{}_{i+n2^{-(l+1)}}^{k} \tag{5.87 b}$$

deoarece $\overset{(l)}{\overline{f_i^{k+2^l}}} = \overset{(l)}{\overline{f_i^k}}$ din ecuația (5.84 a), și

$$\omega_{2^{l+1}}^{k+2^l} = \omega_{2^{l+1}}^{2^l} \cdot \omega_{2^{l+1}}^{k} = \omega_2\, \omega_{2^{l+1}}^{k} = -\,\omega_{2^{l+1}}^{k} \tag{5.87 c}$$

La ultimul pas al ecuației (5.87 c) am înlocuit factorul comun 2^l din indicele și exponentul primului factor, și am folosit faptul că $\omega_2 = -1$.

În concluzie, varianta Cooley-Tukey (1965) a algoritmului FFT este:

$$\left.\begin{aligned} \overset{(l+1)}{f_i^k} &= \overset{(l)}{f_i^k} + \omega_{2^{l+1}}^{k}\, \overset{(l)}{f^k_{i+n2^{-(l+1)}}} && (5.88\ a)\\ \overset{(l+1)}{f_i^{k+2^l}} &= \overset{(l)}{f_i^k} - \omega_{2^{l+1}}^{k}\, \overset{(l)}{f^k_{i+n2^{-(l+1)}}} && (5.88\ b) \end{aligned}\right\}\ \text{pentru } l = 0, 1, \ldots, q-1$$

unde $k = 0, 1, \ldots, 2l-1$ și $i = 0, 1, \ldots, n2^{-(l+1)} - 1$ cu condiția de start $\overset{(0)}{f_i^0} = f_i$. Rezultatul transformatei se obține cu:

$$\bar{f}^k = \frac{1}{n}\overset{(q)}{f_0^k} \tag{5.88 c}$$

Dacă $\overset{(l+1)}{f_i^k}$ se scrie în memorie peste $\overset{(l)}{f_i^k}$, iar $\overset{(l+1)}{f_i^{k+2^l}}$ peste $\overset{(l)}{f^k_{i+n2^{-(l+2)}}}$ atunci nu este necesar spațiu de memorie suplimentar. Armonicele finale se obțin în ordinea binară inversă. Dacă $k = k_{q-1}2^{q-1} + k_{q-2}2^{q-2} + \ldots + k_1 2 + k_0$ unde k_p este cifra de pe poziția p în reprezentația binară a lui k, atunci această armonică se va afla în locația $\overset{(q)}{f^{k'}}$ unde $k' = k_0 2^{q-1} + k_1 2^{q-2} + \ldots + k_{q-2} 2 + k_{q-2}$. Dacă analiza armonică trebuie urmată de o sinteză care inversează pașii de mai sus, atunci nu mai este necesară sortarea armonicelor în ordinea naturală. Acest caz apare în rezolvarea problemelor de cîmp cu metoda convoluției. Dacă armonicele sînt date de ieșire, se va impune o sortare la sfîrșitul execuției algoritmului de bază. Această sortare e cunoscută de obicei cu numele de inversare binară (bit-reversal), deoarece armonicele sînt produse în această ordine. Altfel, dacă se elimină scrierea în aceleași locații de memorie prin depunerea rezultatului într-un alt masiv, sortarea poate fi executată la fiecare nivel, așa cum se arată în figura 5.12.

Transformata Fourier rapidă a ecuațiilor (5.88) poate fi inversată prin rezolvarea la nivelul l în termenii nivelului $l + 1$, obținîndu-se transformata inversă:

$$\left.\begin{aligned} \overset{(l)}{f_i^k} &= \frac{1}{2}\left(\overset{(l+1)}{f_i^k} + \overset{(l+1)}{f_i^{k+2^l}}\right) && (5.89\ a)\\ \overset{(l)}{f^k_{i+n2^{(l+1)}}} &= \frac{1}{2}\,\omega_{2^{l+1}}^{-k}\left(\overset{(l+1)}{f_i^k} - \overset{(l+1)}{f_i^{k+2^l}}\right) && (5.89\ b) \end{aligned}\right|\ \text{pentru } l = q-1, q-2, \ldots, 0$$

unde $k = 0, 1, \ldots, 2l-1$ și $i = 0, 1, \ldots, n2^{-(l+1)} - 1$ cu condiția de start:

$$\overset{(q)}{f_0^k} = n\bar{f}^k \tag{5.89 c}$$

Rezultatele sintetice se găsesc din

$$f_i = \overset{(0)}{\bar{f}}{}^{0}_{i}, \ i = 0, 1, \ldots, n-1 \qquad (5.89\ d)$$

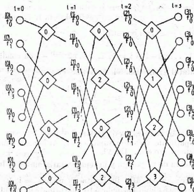


Fig. 5.12 Diagrama de circulație a datelor pentru varianta Cooley-Tukey a transformatei Fourier rapide. Căsuțele evaluează recurența (5.88). Numărul din căsuță este puterea rădăcinii n a unității, folosită ca un factor de fază. (După Temperton 1977).

O analiză a ecuațiilor (5.89) arată că rezultatul final nu se modifică dacă factorul n este eliminat din ecuațiile (5.89 c), iar factorii de 1/2 sînt eliminați din ecuațiile (5.89 a) și (5.89 b). Aceasta echivalează cu introducerea factorului 2^{-1} în membrul drept al definiției transformatei parțiale (5.48 a).

Deoarece o transformată inversă poate fi scrisă ca o transformată directă prin înlocuirea lui w cu w^{-1}, schimbarea rolurilor între f și $\bar{f}$ și prin înmulțirea cu n^{-1} în mod convenabil, ecuațiile (5.89) mai asigură o formulare alternativă a transformatei directe:

$$\left.\begin{aligned} \overset{(l)}{\bar{f}}{}^{k}_{i} &= \overset{(l+1)}{\bar{f}}{}^{k}_{i} + \overset{(l+1)}{\bar{f}}{}^{k+2^{l}}_{i} \qquad (5.90\ a) \\ \overset{(l)}{\bar{f}}_{i+n2^{-(l+1)}} &= \omega^{k}_{2^{l+1}}\left(\overset{(l+1)}{\bar{f}}{}^{k}_{i} - \overset{(l+1)}{\bar{f}}{}^{k+2^{l}}_{i}\right) \qquad (5.90\ b) \end{aligned}\right\} \text{pentru } l = q-1, q-2, \ldots, 0$$

unde $k = 0, 1, \ldots, 2^{l}-1$ și $i = 0, 1, \ldots, n2^{-(l+1)}-1$ cu valoare de start

$$\overset{(q)}{\bar{f}}{}^{k}_{0} = f_k \qquad (5.90\ c)$$

şi rezultatele

$$f^{l} = n^{-1} \overset{(0)}{f_l^0} \qquad (5.90\ d)$$

Aceasta este varianta Gentleman-Sande (1966) a transformatei Fourier rapide.

FFT se mai poate baza pe recurenţele (5.88) sau (5.90). Implementările mai pot diferi prin modul de memorare a transformatelor parţiale intermediare. De exemplu Ubrich (1969) foloseşte ecuaţiile (5.88), iar Pease (1968) ecuaţile (5.90).

5.5.3 Vectorizarea

Tehnicile pentru implementarea FFT pe calculatoare vectoriale pipeline pot fi ilustrate de codul **FORTRAN** pentru algoritmul Cooley-Tukey (ecuaţiile 5.88), prezentat în fig. 5.13. Acest program constă dintr-o rutină de control (în partea superioară) care apelează subrutina **RECUR** pentru evaluarea recurenţei (5.88). **RECUR** (**F**, **G**, **W**, L, N) execută odată recurenţa pentru L = 1 asupra datelor de intrare furnizate de vectorul complex **F** şi plasează rezultatele în vectorul complex **G**. Vectorul W conţine puteri ale rădăcinii n a unităţii, iar N = n este numărul total al variabilelor complexe. La nivele succesive intrările şi ieşirile alternează între masivele **F** şi **G**, de aceea apelurile la **RECUR** se fac în pereche. Deoarece vectorii **F** şi **G** sînt diferiţi, ieşirile nu se scriu în spaţiul pentru intrări. Aceasta determină menţinerea armonicelor în ordine naturală şi permite celor mai multe compilatoare să vectorizeze instrucţiunile buclei **DO I1**. Presupunem că W a fost încărcat înainte cu puteri ale rădăcinii n a unităţii astfel încît

$$\omega_n^k \text{ este memorat în } \mathbf{W}(k+1) \qquad (5.91\ a)$$

iar transformatele parţiale sînt aranjate astfel încît

$$\overset{(l)}{f_i^k} \text{ este memorat în } \mathbf{F} \text{ sau } \mathbf{G}\ (2^l i + k + 1) \qquad (5.91\ b)$$

Celelalte variabile **FORTRAN** sînt L1 = l + 1, K1 = k + 1, I1 = = i + 1, I2L = 2^l, N2L1 = $n2^{-(l+1)}$, I2L2 = 2^{l+1}, N2L2ξ = $n2^{-(l+2)}$, LOG2N = $\log_2 n$. Instrucţiunile din interiorul buclei **DO 20** realizează implementarea recurenţei Cooley-Tukey (5.88). Ele sînt scrise cu indecşii explicit în termenii variabilelor de control ale buclei pentru a ilustra posibilitatea de vectorizare.

În forma scrisă mai sus, cele mai multe compilatoare vor înlocui bucla **DO I1.** cu instrucţiuni vectoriale. Deoarece W nu este o funcţie de I1, va fi înlocuit de scalarul Wk în bucla interioară care poate fi implementată cu instrucţiuni de forma

vector = vector + scalar * vector

Lungimea vectorului este $n2^{-(l+1)}$ şi descreşte la fiecare apel al subrutinei **RECUR**, luînd valorile n/2, n/4, ..., 4, 2, 1. Intervalul între locaţiile de memorie ale elementelor succesive ale vectorului **F**, folosite în bucla **DO 11**, este 2^l şi ia valorile 1, 2, 4, 8, 16, ..., n/2. În consecinţă, în ultimele faze ale algoritmului vor exista conflicte de acces la memorie pe calculatoare

```
C     MAIN CONTROL PROGRAM
       N = 2**LOG2N
    DO 10 L1 = 1, LOG2N, 2
      CALL RECUR (F, G, W, L1-1, N)
      IF (L1.EQ.LOG2N) STOP
  10   CALL RECUR (G, F, W, L1, N)
   STOP

  SUBROUTINE RECUR(F, G, W, L, N)
  COMPLEX F(N), G(N), W(N), V(N), WK
      I2L = 2**L
      I2L1 = 2**I2L
      N2L1 = N/I2L1
      INC1 = I2L
      IF (L.GE.2) INC1 = INC1+1
      INC2 = I2L1
      IF (L+1.GE.2) INC2 = INC1
      INC4 = I2L-INC2
  DO 20 K1 =1, I2L
     WK = W(N2L1*K1-N2L1+1)
  DO 20 I1 1,N2L1
        V(I1) = WK*F(INC1*I1+K1+INC3)
        G(INC2*I1+K1-INC2) = F(INC1*I1+K1-INC1) + V(I1)
 20   G(INC2*I1+K1+INC4) = F(INC1*I1+K1-INC1) - V(I1)
   RETURN
   END
```

Fig. 5.13 Programul de control şi subrutina RECUR pentru schema A cu vectorizarea buclei DO I1

cu un număr de blocuri de memorie putere a lui 2. De exemplu, pe **CRAY-1** vor exista conflicte pentru $l \geqslant 2$ sau intervale multiple de 4 numere complexe sau 8 numere reale.

Aceste conflicte pot fi eliminate cu costul unui mic spaţiu de memorie suplimentar, prin adăugarea unităţii la intervalele de memorare INC1 şi INC2 pentru $l \geqslant 2$. Spaţiul de memorie necesar creşte cu 1/4, iar intervalele între locaţiile de memorie ale elementelor succesive ale vectorului **F** devin 1, 2, 5, 9, 17, ..., $n/2 + 1$ la nivelele $l = 0, 1, 2, 3, 4, \ldots, \log_2(n) - 1$. Această modificare în modul de memorare are loc în instrucţiunea **IF** tocmai înainte de bucla **DO 20**.

În bucla **DO 20** sînt 10 operaţii aritmetice reale scalare, deci pe un calculator caracterizat de un $n_{1/2}$, timpul de execuţie al acestui algoritm (pe care îl denumim schema A) este proporţional cu:

$$t_A \doteq \sum_{l=0}^{\log_2(n)-1} 10(n_{1/2} + n \cdot 2^{-(l+1)}) 2^l \quad (5.92\ a)$$

$$= 10\, n_{1/2} \sum_{l=0}^{\log_2(n)-1} 2^l + 5n \sum_{l=0}^{\log_2(n)-1} 1 \quad (5.92\ b)$$

$$= 10\, n_{1/2}(n - 1) + 5n \log_2 n \quad (5.92\ c)$$

Ultimul termen al ecuației (5.92 c) este numărul obișnuit de operații seriale pentru algoritmul FFT (cînd $n_{1/2} = 0$), iar primul termen oglindește efectul paralelismului hardware prin intermediul valorii lui $n_{1/2}$.

Este evident că buclele **DO K1** și **DO I1** pot fi inter-schimbate fără a modifica efectul instrucțiunilor buclei **DO 20**. Dacă se face aceasta, cum se

```
      SUBROUTINE RECUR (F, G, W, L, N)
      COMPLEX F(N), G(N), V(N), W(N)
            I2L = 2**L
            I2L1 = 2*I2L
            N2L1 = N/I2L1

            INC1 = I2L
            IF (L.GE.2) INC1 = INC1+1
            INC2 = I2L1
            IF (L+1.GE.2) INC2 = INC2+1
            INC3 = N2L1*INC1-INC1
            INC4 = I2L-INC2

      DO 20 I1 = 1, N2L1
      DO 20 K1 = 1, I2L
            V(K1) = W(N2L1*K1-N2L1+1)*F(INC1*I1+K1+INC3)
            G(INC2*I1+K1-INC2) = F(INC1*I1+K1-INC1) + V(K1)
20          G(INC2*I1+K1+INC4) = F(INC1*I1+K1-INC1) - V(K1)
      RETURN
      END
```

Fig. 5.14 Subrutina RECUR pentru schema B cu verctorizarea buclei DO K1

arată în fig. 5.14, bucla **DO K1** devine cea mai interioară și poate fi înlocuită cu instrucțiuni vectoriale. Lungimea vectorului este 2^l sau 1, 2, 4, ..., n/2 la treceri succesive, iar intervalul între locațiile de memorie este $n2^{-(l+1)}$ sau n/2, n/4, ..., 4, 2, 1. Conflictele de acces la memorie sînt o problemă serioasă în fazele timpurii ale execuției acestui algoritm, dar pot fi evitate în maniera descrisă anterior. Acum variabila **W** este un vector în bucla interioară, iar înmulțirea este o operație vector * vector. Timpul de execuție al acestui algoritm alternativ (schema B) este proporțional cu

$$t_B = \sum_{l=0}^{\log_2(n)-1} 10(n_{1/2} + 2^l)\, n2^{-(l+1)} \qquad (5.93\ a)$$

$$= 10\, n_{1/2} \sum_{l=0}^{\log_2(n)-1} n \cdot 2^{-(l+1)} + 5n \sum_{l=0}^{\log_2(n)-1} 1 \qquad (5.93\ b)$$

cu $l' = \log_2(n) - 1 - l$ și inversînd ordinea primei sume se obține

$$t_B = 10\, n_{1/2} \sum_{l=0}^{\log_2(n)-1} 2^{l'} + 5n \log_2 n \qquad (5.93\ c)$$

$$= 10\, n_{1/2}(n-1) + 5n \log_2 n \qquad (5.93\ d)$$

$$= t_A \qquad (5.93\ e)$$

Deci, cele două scheme se execută în acelaşi interval de timp. Totuşi au caracteristici diferite. Schema **A** începe prelucrările cu vectori lungi, care se reduc în dimensiune odată cu avansarea procesului de calcul, în timp ce schema **B** începe cu vectori mici care cresc în lungime cu avansarea procesului de calcul. Performanţa tuturor calculatoarelor paralele creşte odată cu creşterea lungimii vectorului, de unde sugestia pentru un nou algoritm. Se execută primele p nivele ale algoritmului FFT cu schema **A**, iar ultimele q—p cu schema **B**. Acest algoritm combinat se execută într-un timp proporţional cu

$$t_{A+B} = 10\, n_{1/2} \left(\sum_{l=0}^{p-1} 2^l + \sum_{l=p}^{\log_2(n)-1} n \cdot 2^{-(l+1)} \right) + 5n \log_2 n \qquad (5.94\ a)$$

$$= 10\, n_{1/2}(2^p + n \cdot 2^{-p} - 2) + 5n \log_2 n \qquad (5.94\ b)$$

Prin derivarea ecuaţiei (5.94 b) în raport cu p, se găseşte condiţia de minim a timpului de execuţie.

$$\frac{dt_{A+B}}{dp} = 10\, n_{1/2} \log_e 2(2^p - n2^{-p}) = 0 \qquad (5.95\ a)$$

Deci

$$2^p = n/2^p,\ 2^{2p} = n \qquad (5.95\ b)$$

şi

$$p = \frac{1}{2} \log_2 n \qquad (5.95\ c)$$

Cu acest p optim, lungimea vectorului minim este $2^p = \sqrt{n}\left(\text{sau } \sqrt{\frac{1}{2}\, n}\right.$ dacă n nu este un multiplu de $\left.4\right)$, iar timpul de execuţie (presupunînd că n este un multiplu de 4) este

$$t_{A+B} = 20\, n_{1/2}(\sqrt{n} - 1) + 5\, n \log_2 n \qquad (5.96)$$

Acest tip de algoritm combinat a fost dezvoltat în mod independent de Roberts (1977) şi Temperton (1979 b). Versiunea propusă de Temperton se bazează pe recurenţa Gentleman-Sande (5.90) şi reprezintă baza subrutinei **CFFT2** scrisă în limbaj de asamblare **CAL** de Petersen (1978) pentru biblioteca de programe ştiinţifice a calculatorului **CRAY X-MP**.

Orice algoritm eficient mai incorporează două simplificări. Acestea se referă la valoarea multiplicatorilor $\omega_2^k l + 1 = \omega_n^{kn/2l+1}$, care intervin atît recurenţa Cooley-Tukey cît şi în recurenţele Gentleman-Sande. Cînd $k = 0$ multiplicatorul este unitatea, iar operaţia de înmulţire poate fi eliminată prin scrierea unei bucle separate pentru acest caz. La calculul primului

nivel al transformatelor parţiale, cînd $l = 0$ în recurenţele (5.88), această situaţie intervine la fiecare evaluare a ecuaţiilor (5.88). La alte nivele, intervine la prima utilizare a ecuaţiilor. La calculul celui de-al doilea nivel al transformatelor parţiale, pentru $l = 1$ în ecuaţiile (5.88), $k = 0, 1$ şi multiplicatorii sînt 1 respectiv i. Deoarece înmulţirea cu i nu face decît să schimbe părţile imaginară şi reală şi modifică semnul părţii imaginare, nu este necesară nici o înmulţire pentru calculul transformatelor la al doilea nivel. Din nou se foloseşte o buclă separată şi pentru acest caz.

Dacă numărul elementelor n ce sînt transformate conţin factori de 4, se pot reduce mai mult operaţiile necesare (Singleton 1969). Devine avantajos să se combine două aplicaţii ale recurenţei (5.88) într-o singură recurenţă implicînd 4 valori de intrare $\overset{(l-1)}{\overline{f_i^k}}$ şi 4 valori de ieşire $\overset{(l+1)}{\overline{f_i^k}}$. Dacă presupunem că n este o putere a lui 4 şi deci $\log_2 n$ este par, se poate calcula transformata cu recurenţa:

$$\text{pentru } l = 1, 3, 5, \ldots, \log_2(n) - 1$$

$$\text{pentru } k = 0, 1, \ldots, \frac{1}{2}\, 2^l - 1;\; i = 0, 1, \ldots, n2^{-(l+1)}$$

$$a = \omega_{2^{l+1}}^{k} \overset{(l-1)}{\overline{f^k_{i+\frac{1}{2}n2^{-l}}}},\; b = \omega_{2^{l+1}}^{2k} \overset{(l-1)}{\overline{f^k_{i+n2^{-l}}}},\; c = \omega_{2^{l+1}}^{3k} \overset{(l-1)}{\overline{f^k_{i+\frac{3}{2}n2^{-l}}}} \tag{5.97 a}$$

$$d = \overset{(l-1)}{\overline{f_i^k}} + b,\; g = a + c \tag{5.97 b}$$

$$e = \overset{(l-1)}{\overline{f_i^k}} - b,\; h = a - c \tag{5.97 c}$$

$$\overset{(l+1)}{\overline{f_i^k}} = d + g,\; \overset{(l+1)}{\overline{f_i^{k+\frac{1}{2}2^l}}} = e + i\,h \tag{5.97 d}$$

$$\overset{(l+1)}{\overline{f_i^{k+2^l}}} = d - g,\; \overset{(l+1)}{\overline{f_i^{k+\frac{3}{2}2^l}}} = e - ih \tag{5.97 e}$$

Evoluţia procesului de calcul pentru cazul în care n este o putere a lui 4 ($n = 4^2$) este prezentată în fig. 5.15. Evaluarea recurenţei (5.97) implică trei înmulţiri pentru a-i calcula pe a, b şi c şi 8 adunări complexe, conform ecuaţiilor (5.97 b, c, d, e). Deci, 34 operaţii reale pentru (1/2) $\log_2 n$ valori ale lui l şi n/4 combinaţii ale lui k şi i. Rezultă un total de $4{,}25\, n \log_2 n$ operaţii reale, în comparaţie cu $5 \log_2 n$ pentru transformata corespunzătoare unei puteri a lui 2, sau o economie de 15%. Dacă $l = 1$, $k = 0$, toţi multiplicatorii sînt unitatea şi, ca şi în cazul transformatei putere a lui 2, se justifică programul suplimentar pentru a se elimina operaţiile nenecesare. Transformatele corespunzătoare lui n putere a lui 2

şi a lui 4 pot fi combinate uşor pentru a se realiza o transformată eficientă pentru orice putere a lui 2, care elimină din n întii factorii de 4 şi apoi, dacă este necesar, un factor de 2.

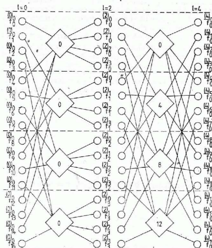


Fig. 5.15 Evoluţia procesului de calcul corespunzător transformatei cu n=16. Căsuţele reprezintă evaluarea recurenţei (5.97). Numărul din căsuţă este puterea celei de-a n-a rădăcină a unităţii folosită la evaluarea constantei a.

5.5.4 Implementarea paralelă

Pentru masivele de procesoare se pot obţine performanţe ridicate dacă paralelismul algoritmului este făcut să corespundă paralelismului hardware al masivului, adică numărului de procesoare masiv. Paralelismul algoritmilor analizaţi în § 5.5.3. variază de la $n/2$ la $\sqrt{n}$ şi deci nu sînt foarte convenabili pentru implementare pe masive de procesoare, care au un număr fix de procesoare, deci paralelismul este fixat. Totuşi, dacă sînt combinate sub forma unei singure bucle, buclele **DO K1** şi **DO I1** din figura 5.13 sau 5.14, toate operaţiile aritmetice pot fi executate cu un paralelism fixat n. Dacă acesta corespunde masivului (de exemplu, transformarea a 4096 valori pe un calculator **ICL DAP 64×64**), se atinge idealul.

Vom numi algoritmul de mai sus **PARAFT** deoarece este cel mai indicat pentru execuţia pe paracalculator. În fig. 5.16 se prezintă detalii ale subrutinei **RECUR**. Bucla **DO 20** nu conţine operaţii aritmetice, ci constă numai în transferuri de date necesare pentru pregătirea operaţiilor ce se vor executa în bucla **DO 30**. Masivele complexe **U** şi **E** sînt copii ale

```
      SUBROUTINE RECUR (F, G, W, L, N)
      COMPLEX F(N), G(N), W(N), U(N), E(N)

            I2L = 2**L
            I2L1 = 2*I2L
            N2L1 = N/I2L1

            INC1 = I2L
            INC2 = I2L1
            INC3 = N2L1*INC1-INC1
            INC4 = I2L - INC2
      DO 20 K1 = 1, I2L
      DO 20 I1 = 1,N2L1
         U(INC2*I1+K1-INC2) = F(INC1*I1+K1-INC1)
         U(INC2*I1+K1+INC4) = F(INC1*I1+K1-INC1)
         E(INC2*I1+K1-INC2) = F(INC1*I1+K1+INC3)
         E(INC2*I1+K1+INC4) = -F(INC1*I1+K1+INC3)
         V(INC2*I1+K1-INC2) = W(N2L1*K1-N2L1+1)
20       V(INC2*I1+K1+INC4) = W(N2L1*K1-N2L1+1)
      DO 30  J + 1, N
         E(J) = V(J)*E(J)
30       G(J) = U(J)+E(J)
      RETURN
      END
```

Fig. 5.16. Subrutina RECUR pentru PARAFT cu lungimea vectorului N în bucla DO 30.

masivului **F** cu o modificare de semn ce corespunde semnului minus din ecuaţia (5.88 b). Masivul complex **V** conţine copii ale multiplicatorilor în poziţiile corecte astfel încît cele n înmulţiri ale ecuaţiilor (5.88) să poată fi executate în paralel cu prima instrucţiune din bucla **DO 30**. A doua instrucţiune din bucla **DO 30** execută apoi cele n sume ale ecuaţiilor (5.88) în paralel.

Cum s-a menţionat în § 5.5.3 în primele două apeluri ale subrutinei **RECUR** cu $l = 0$ şi $l = 1$ nu se execută înmulţiri. Într-un program eficient se va implementa cod pentru aceste cazuri, ca şi pentru transformate corespunzătoare puterii lui 4. Cînd se evaluează timpul de execuţie al algoritmului, ar trebui considerate şi operaţiile de transfer din interiorul buclei **DO 20**. În cazul masivelor de procesoare, unde operaţiile aritmetice sînt mai lente în comparaţie cu cele de transfer, cum e cazul la **ICL DAP**, bucla **DO 20** nu va reprezenta prea mult din timpul de execuţie. De exemplu, cînd se execută 1024 transformate complexe pe un calculator **ICL DAP 32** $\times$ **32**, operaţiile de transfer reprezintă 10—20% din timpul total (Flanders et al 1977). Pentru masive cu procesoare mai puternice, bucla

DO 20 va avea o importanţă mai mare. Numeroase masive de procesoare au conexiuni şi circuite speciale pentru transferuri, tocmai pentru execuţia lor eficientă în cazul transformatei Fourier rapide. Goodyear **STARAN** şi Burroughs **BSP** sînt două exemple (vezi capitolul 3).

Analiză fig. 5.16 arată că algoritmul **PARAFT** necesită o înmulţire complexă şi o adunare complexă cu paralelism n, la fiecare nivel. Acestea sînt echivalente cu 8 operaţii reale cu paralelism n la $\log_2 n$ nivele sau, dacă se consideră că operaţiile de transfer sînt neimportante, timpul de execuţie este proporţional cu

$$t_{PARAFT} = 8(n_{1/2} + n)\log_2 n \qquad (5.98\text{ a})$$

Comparînd ecuaţia (5.98 a) cu timpul pentru schema **A + B** din ecuaţia (5.96) găsim ca **PARAFT** are o performanţă mai bună decît **A+B** dacă

$$n_{1/2} > 3n\log_2 n/[20(\sqrt{n}-1)-8\log_2 n] \qquad (5.98\text{ b})$$

În fig. 5.17 se prezintă, pe această bază, zonele din planul $(n, n_{1/2}^{-1})$ care avantajează fiecare din cei doi algoritmi cu (m = 1). Consideraţiile de mai sus sînt valabile pentru calculatoare pipeline şi masive de procesoare într-un sens larg (vezi § 1.3.3). Există şi alte alternative, iar Jesshope

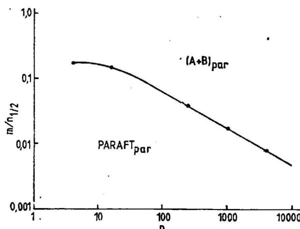


Fig. 5.17 Regiunile planului $(n, m/n_{1/2})$ unde fie schema A+B, fie algoritmul PARAFT au performanţa cea mai bună, cînd se aplică în paralel celor m transformate Fourier de lungime n. Cînd m = 1 se poate alege cel mai bun algoritm pentru o singură transformată

(1980 a) a arătat că o combinaţie a celor două metode este avantajoasă pe masive de procesoare cînd numărul n depăşeşte dimensiunea masivului. În acest caz n este înjumătăţit succesiv cu ajutorul schemei **A + B**, pînă

ce lungimea transferului este inferioară dimensiunii masivului. La acest nivel se folosește algortmul **PARAFT** pentru finalizarea transformatei.

Apoi, dacă trebuie calculate m transformate fiecare de lungime n, soluția va depinde de gradul de paralelism dorit. Se vor alege la început metodele cu un paralelism maxim și se va compara performanța schemei $A + B$ și **PARAFT** în cazul aplicării simultane celor m transformate. Aceste metode sînt utile pentru mașini cu un paralelism natural ridicat, ca **CYBER 205** și **ICL DAP**.

Timpii de execuție vor fi proporționali cu

$$t_{(A+B)_{par}} = 20\, n_{1/2} (\sqrt{n}-1) + 5\, m \cdot n \log_2 n \qquad (5.99\ a)$$

$$t_{PARAFT_{par}} = 8\, (n_{1/2} + m \cdot n) \log_2 n \qquad (5.99\ b)$$

Comparînd cu ecuația (5.4 b) vedem că lungimea vectorului mediu este:

$$\bar{n}_{(A+B)par} = \frac{m \cdot n \cdot \log_2 n}{4(\sqrt{n}-1)} \qquad (5.99\ c)$$

$$\bar{n}_{PARAFT_{par}} = m \cdot n \qquad (5.99\ d)$$

Ecuațiile (5.99) arată că algoritmul (A+B) par are o performanță mai bună cînd

$$\frac{m}{n_{1/2}} > \frac{20(\sqrt{n}-1) - 8 \log_2 n}{3n \log_2 n} \qquad (5.99\ e)$$

Regiunile planului $(n, m/n_{1/2})$ ce avantajează cele două metode sînt prezentate în fig. 5.17. Se observă că schema $(A + B)_{par}$ este favorizată cînd fie numărul transformatelor, fie lungimea lor devin mari.

Dacă, altfel, este avantajoasă limitarea paralelismului algoritmului la aproximativ n sau m, ne putem întreba dacă este avantajos de a repeta cel mai bun algoritm pentru o singură transformată de m ori, sau să alegem cel mai bun algoritm serial și să-l executăm în paralel pentru cele m sisteme (algoritmul **MULTFT**). Algoritmul serial cel mai bun este fie schema A, fie schema B cu $n_{1/2} = 0$, caracterizați de $5n\log_2 n$ operații reale scalare. Algoritmul **MULTFT** are lungimea vectorului mediu $\bar{n}_{MULTFT} = m$ și se va executa într-un timp proporțional cu:

$$t_{MULTFT} = 5n(n_{1/2} + m) \log_2 n \qquad (5.100\ a)$$

Cînd se aplică secvenţial de m ori, celor m sisteme diferite, ceilalţi algoritmi se vor executa într-un timp proporţional cu

$$t_{(A+B)_{seq}} = m\,[20\,m n_{1/2}\,(\sqrt{n}-1) + 5\,n \log_2 n] \tag{5.100 b}$$

$$t_{PARAFT_{seq}} = 8\,m\,(n_{1/2} + n) \log_2 n \tag{5.100 c}$$

cu lungimile vectorului mediu

$$\bar{n}_{(A+B)_{seq}} = \frac{n \log_2 n}{4(\sqrt{n}-1)}, \quad \bar{n}_{PARAFT_{seq}} = n \tag{5.100 d}$$

Deci, algoritmul **MULTFT** va avea o performanţă superioară schemei $(\mathbf{A}+\mathbf{B})_{seq}$ dacă

$$m > \frac{n}{4}\,\frac{\log_2 n}{(\sqrt{n}-1)} \tag{5.101 a}$$

şi va avea o performanţă superioară algoritmului $\mathbf{PARAFT}_{seq}$, dacă

$$\frac{m}{n_{1/2}} > \frac{0{,}625\,n/n_{1/2}}{1 + 0{,}375\,n/n_{1/2}} \tag{5.101 b}$$

Curba plină din figura 5.18 este o reprezentare a ecuaţiei (5.101 a) şi arată regiunea din planul (n, m) unde fie **MULTFT**, fie schema $(\mathbf{A}+\mathbf{B})_{seq}$

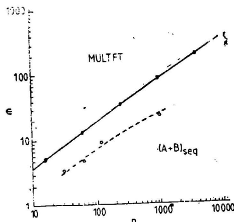


Fig. 5.18 Alegerea celui mai bun algoritm pentru calculul a m transformate Fourier rapide, fiecare de lungime n, pe un calculator cu paralelism natural aproximativ m sau n, pentru care schema A+B este cea mai bună metodă de calcul al unei singure transformate (vezi fig. 5.17). Cel mai bun FFT serial se aplică în paralel transformatelor, MULTFT sau schema A+B se aplică secvenţial, $(A+B)_{seq}$. În acest caz o singură curbă este valabilă pentru toate valorile lui $n_{1/2}$ şi împarte planul (n, m) în regiuni unde fie MULTFT, fie schema $(A+B)_{seq}$ au performanţa cea mai bună. Cerculeţele provin din rezultatele măsurătorilor lui Temperton (1979b) pe CRAY-1.

are o performanţă superioară. Observăm că ecuaţia (5.101 a) este independentă de $n_{1/2}$ şi o singură reprezentare grafică se aplică tuturor calculatoarelor. Deci, în sens larg, dacă numărul sistemelor ce urmează să fie transformate depăşeşte 1/10 din lungimea sistemelor, atunci este avanta-

jos să folosim cel mai bun algoritm serial pentru toate sistemele în paralel, decît cel mai bun algoritm paralel pentru fiecare sistem în secvență. Temperton (1979 b) a comparat performanța a 2 programe **FORTRAN** pe **CRAY-1** pentru algoritmi comparabili cu **MULTFT** și schema $(\mathbf{A} + \mathbf{B})_{seq}$. Măsurătorile sale sînt prezentate pe aceeași figură cu cerculețe și linie punctată. Aceste măsurători concordă cu rezultatele teoretice prezentate anterior și vor fi în concordanță perfectă dacă viteza anticipată a algoritmului **MULTFT** relativă la cea a schemei $(\mathbf{A} + \mathbf{B})_{seq}$ va crește de un număr între 2 și 3. Acest rezultat este de așteptat deoarece algoritmul **MULTFT** are o indexare mult mai simplă decît schema $(\mathbf{A}+\mathbf{B})_{seq}$ (incrementul buclei vectoriale cea mai interioară pentru cele m sisteme este întotdeauna unitatea) și de aceea memoria este accesată în modul cel mai convenabil.

Fig. 5.19 ilustrează o comparație similară între **MULTFT** și aplicarea secvențială a algoritmului **PARAFT**. Se folosește cînd, pe baza fig. 5.17, se deduce că algoritmul **PARAFT** este cel mai bun pentru execuția unei singure transformate. Pot fi folosite următoarele valori limită :

$$m = \begin{cases} 0{,}625\ n, & n \ll n_{1/2} \quad (5.102\ a) \\ 0{,}454\ n_{1/2}, & n = n_{1/2} \quad (5.102\ b) \end{cases}$$

iar pentru n mare se atinge o valoare limită a lui m

$$m = 1{,}67\ n_{1/2},\ n \gg n_{1/2} \quad (5.102\ c)$$

5.5.5 Considerații privind operațiile de transfer

Comparațiile performanțelor efectuate mai sus se bazează pe presupunerea că întîrzierile datorate operațiilor de transfer nu influențează semnificativ timpul de execuție (vezi § 5.1.4).

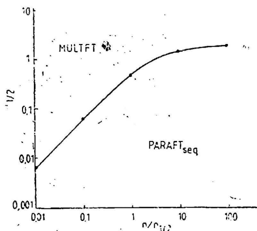


Fig. 5.19 Alegerea celui mai bun algoritm pentru calculul a m transformate Fourier rapide, fiecare de lungime n, pe un calculator cu paralelismul natural de aproximativ n sau m și pentru care PARAFT este cea mai bună metodă de calcul al unei singure transformate (vezi fig. 5.17). PARAFT se aplică secvențial celor m transformate, $PARAFT_{seq}$. Curba separă în planul $(n/n_{1/2})$ regiunile în care fie MULTFT, fie $PARAFT_{seq}$ au o performanță mai bună.

Astfel de întîrzieri influenţează performanţa masivelor de procesoare pentru care **PARAFT** este probabil cel mai bun din punct de vedere al operaţiilor aritmetice (vezi fig. 5.17). De aceea, vom aborda problema transferului pentru acest algoritm. Evident întîrzierea datorată transferurilor va depinde de topologia de interconectare a procesoarelor din masiv. Este motivul pentru care vom considera o clasă generală de astfel de conexiuni, care include cele mai multe din topologiile calculatoarelor comercializate. Analiza ce urmează foloseşte pe cea a lui Jesshope ((1980 a) care a întreprins o analiză comprehensivă a implementării transformatelor rapide pe masive de procesoare. Alte rezultate privind operaţiile de transfer şi transpuneri pe masive de procesoare sînt prezentate în lucrările Jesshope (1980 b, c). În plus, Nassimi şi Sahni (1980) au abordat implementarea topologiilor amestecului perfect şi inversarea biţilor.

Să considerăm un masiv de procesoare cu P procesoare, aranjate ca un masiv cartezian k dimensional, cu Q procesoare pe fiecare direcţie :

$$P = Q^k \qquad (5.103\ a)$$

şi, conform practicii curente, vom lua Q o putere a lui 2.

$$Q = 2^q,\ q = 1, 2, \ldots \qquad (5.103\ b)$$

Fiecare procesor accesează memoria sa proprie şi cea a vecinilor mediaţi după fiecare direcţie. Observăm, în particular, că nu există conexiuni după direcţii diagonale. Pentru $q \geqslant 2$ această topologie implică 2k conexiuni la fiecare procesor şi kP căi de date pentru întregul masiv, dacă presupunem că o cale este bidirecţională. Pentru $q = 1$ numerele de mai sus se înjumătăţesc. La marginile masivului procesoarele sînt legate în sens periodic după fiecare direcţie. Dacă l este direcţia şi i_l coordonata după direcţial, periodicitatea se referă la faptul că toate coordonatele se interpretează mod Q (mod este prescurtarea pentru modulo), deci,

$$0 \leqslant i_l \leqslant Q-1,\ \ l = 1,\ 2, \ldots, k \qquad (5.104\ a)$$

iar conectivitatea vecinătăţii proxime înseamnă că procesoarele sînt conectate dacă coordonatele lor diferă prin ± 1 numai după o singură direcţie.

Dorim să folosim acest masiv k dimensional pentru memorarea unui masiv uni-dimensional (f_i, $i = 1, \ldots, n$) de date ce urmează să fie transferate. Pentru a evita complicaţii vom presupune că numărul procesoarelor este egal cu cel al datelor. Numerotînd secvenţial procesoarele după prima, a doua, ..., a k-a direcţie, stabilim corespondenţa

$$i = i_1 \pm i_2 Q + i_3 Q^2 + \ldots + i_k Q^{k-1} \qquad (5.104\ b)$$

iar conectivitatea vecinătăţii proxime înseamnă că procesorul i este legat la :

$$(i \pm 1) \bmod Q,\ (i \pm Q) \bmod Q^2, \ldots, (i \pm Q^{k-1}) \bmod Q^k \qquad (5.104\ c)$$

În această clasă intră calculatoarele:
(a) ILLIAC IV $P = 64$, $k = 2$, $Q = 8$, $q = 3$ şi procesorul i se conectează cu $(i \pm 1, i \pm 8)$ mod 64. Observăm că deşi conexiunile interne sînt bi-dimensionale, conexiunile la margine transformă masivul într-o structură periodică uni-dimensională. Din această cauză maşina nu face parte strict din această clasă;
(b) ICL DAP $P = 4096$, $k = 2$, $Q = 64$, $q = 6$ şi procesorul i se conectează cu $(i \pm 1)$ mod 64, $(i \pm 64)$ mod 409;
(c) HYPERCUBE (binar) $P = 16$, $k = 4$, $Q = 2$, $q = 1$ şi procesorul i se conectează cu $(i \pm 1)$ mod 2, $(i \pm 2)$ mod 4, $(i \pm 4)$ mod 8, $(i \pm 8)$ mod 16. Se înţelege că asigurarea a kP căi de date limitează dimensionalitatea masivelor reale la 2 cînd numărul procesoarelor este mare. Cel mai înalt grad de conectivitate îl are Hipercubul (Millard 1975) care are $Q = 2$, 3 sau 4 şi $k = 4$. Calculatoarele ce nu intră în clasa de mai sus sînt cele care asigură conexiuni pe diagonale, cum este CLIP de la University College din Londra (Duff 1978, 1980 a, 1980 b) care asigură conexiuni pe 8 direcţii într-un masiv bi-direcţional. Definiţia noastră permite conexiuni numai la cele mai apropiate 4 procesoare.

Operaţiile de transfer au loc, în cazul algoritmului **PARAFT**, numai în bucla **DO 20** din fig. 5.16. O reordonare parţială se desfăşoară la fiecare nivel. În prima şi a treia instrucţiune, valorile lui F separate prin N2L1 $= n/2^{l+1}$ se deplasează pentru a se aranja sub fiecare alta. O deplasare şi inversare similară au loc în a doua şi a patra instrucţiune. Dacă luăm algoritmul Cooley-Tukey fără reordonare, algoritmul **PARAFT** poate fi simplificat şi reformulat astfel:

Pentru $l = 0, 1, \ldots, \log_2(n)-1$ execută (5.105 a)

(a) Deconectează grupele pare de $n/2^{l+1}$ procesoare şi înmulţeşte cu un masiv memorat anterior de factori de fază.

(b) Transferă toate datele $n/2^{l+1}$ noduri prin reţeaua liniară modulo $2n/2^{l+1}$ şi memorează într-o zonă tampon de lucru. Dacă reţeaua nu posedă conexiuni ciclice modulo $2n/2^{l+1}$, atunci trebuie executat acest pas în doi paşi cu mascare, deplasînd grupele impare şi pare de $n/2^{l+1}$ elemente în direcţii opuse.

(c) Deconectează grupele impare de $n/2^{l+1}$ procesoare şi schimbă semnul spaţiului de lucru.

(d) Adună spaţiul de lucru la masivul de date.

Sfîrşit.

Sub această formă numărul operaţiilor de transfer este

$$n/2,\ n/4,\ n/8,\ \ldots,\ 4,\ 2,\ 1 \qquad (5.105\ b)$$

corespunzător valorilor lui l de la 0 la $\log_2(n)-1$. Dacă cele P procesoare formează un masiv liniar ($k = 1$) atunci numărul total de operaţii de transfer este

$$R^{(1)} = \frac{n}{2} + 2\left(\frac{n}{4} + \frac{n}{8} + \ldots + 2 + 1\right) \qquad (5.106\ a)$$

$$= \frac{n}{2} + 2\left(\frac{n}{2} - 1\right) = \frac{3n}{2} - 2 \qquad (5.106\ b)$$

Primul termen al ecuaţiei (5.106 a) ia în considerare faptul că, pentru $l = 0$, periodicitatea masivului permite execuţia transferului cu o singură deplasare la stînga sau dreapta cu $n/2$ poziţii. Factorul 2 al celui de-al doilea termen ia în considerare că în cadrul pasului (b), grupele pare şi impare se deplasează în urma execuţiei unor operaţii separate.

Dacă masivul este multidiomensional şi, aşa cum am presupus, datele umplu exact masivul de procesoare, atunci prima deplasare relativă a datelor cu $n/2$ elemente vectoriale poate fi executată cu un transfer de $Q/2$ după dimensiunea k, a doua deplasare de $n/4$ cu un transfer de $Q/4$ după dimensiunea k. Se pot continua transferurile succesive cu jumătate din valoarea anterioară, după direcţia k pînă se atinge unitatea. Pentru a reduce la jumătate aceste operaţii trebuie să realizăm transferul acum după dimensiunea $(k-1)$ cu $Q/2$ pînă ce se atinge unitatea. În fig. 5.20 se prezintă un exemplu pentru un masiv de 16×16 procesoare. Evident, procesul de mai sus solicită transferuri diferite de ecuaţia (5.106 a), unde Q va înlocui n pentru fiecare dimensiune. Numărul total de transferuri de numere complexe este în cazul conexiunii k-dimensionale

$$R^{(k)} = k\,(3Q/2-2) \qquad (5.107\ a)$$

Numărul minim de astfel de operaţii de transfer se obţine pentru $Q = 2$, adică pentru un hipercub binar, în care caz

$$R^{(k)} = k = \log_2 p = \log_2 n \qquad (5.107\ b)$$

Numărul operaţiilor de transfer complexe solicitate de alte calculatoare sînt, folosind ecuţia (5.107 a) :

(a) "ILLIAC IV,, ; ('' indică o maşină ipotetică asemănătoare cu ILLIAC IV, care este caracterizată de o conexiune bi-dimensională periodică)

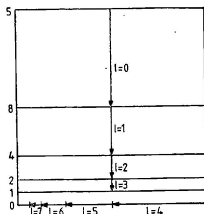


Fig. 5.20 Mărimea şi direcţia transferurilor necesare pentru evaluarea unei transformate a $2^8 = 256$ valori pe un masiv 16×16, $l = 0, 1, \ldots, 7$ este variabila de control a buclei în expresia (5.105 a). Nivelul transformatei parţiale calculată după deplasare este $l+1$.

$$P = n = 64, \quad Q = 8, k = 2$$

$$R^{(2)} = 20 = 3{,}3 \log_2 n \qquad (5.108\ a)$$

(b) **ICL DAP**

$$P = n = 4096, \quad Q = 64, \quad k = 2$$

$$R^{(2)} = 188 = 15{,}7 \log_2 n \qquad (5.108\ b)$$

sau deoarece $k = (\log_2 n)/q$ pentru situaţia $n = P$

$$R^{(k)} = \frac{3Q/2-2}{q} \log_2 n \qquad (5.108\ c)$$

Numărul operaţiilor aritmetice paralele reale necesare după execuţia deplasărilor complexe anterioare este de $8\log_2 n$ [vezi ecuaţia (5.98)]. Prin urma-

re, dacă presupunem că o operaţie paralelă de transfer este de γ ori mai rapidă decît o operaţie aritmetică paralelă, pentru $n = P$

$$\frac{t_R}{t_A} = \frac{\text{timp consumat pentru operaţii de transfer}}{\text{timp consumat pentru operaţii aritmetice}} = \frac{2(3Q/2 - 2)}{8\gamma q} \quad (5.109\text{ a})$$

unde factorul 2 de la numărător semnifică că fiecare transfer complex în ecuaţia (5.108 c) reprezintă deplasarea a două numere reale. Pentru calculatoarele pe care le-am ales, se obţine aproximativ. :

(a) 'ILLIAC IV'

$$\gamma = 2,\ t_R/t_A = 42\% \quad (5.109\text{ b})$$

(b) ICL DAP

$$\gamma = 20,\ t_R/t_A = 19\% \quad (5.109\text{ c})$$

(c) HYPERCUBE

$$\gamma = 10,\ t_R/t_A = 2{,}5\% \quad (5.109\text{ d})$$

Se observă deci că întîrzierile provocate de operaţiile de transfer deşi semnificative, nu domină calculul transformatelor Fourier complexe în cazul masivelor de procesoare considerate şi pot fi ignorate într-o primă estimare a performanţei algoritmilor transformatelor rapide. În § 5.5.6 vom vedea că, pentru calculul transformatelor teoretice numerice pe anumite tipuri de masive de procesoare, situaţia se inversează.

În general, transferurile vor domina operaţiile aritmetice cînd $t_R/t_A \geqslant 1$ sau, reamintind că $q = \log_2 Q$, cînd

$$3Q/2 - 2 \geqslant 4\gamma \log_2 Q \quad (5.110\text{ a})$$

sau pentru Q mare, cînd

$$Q/\log_2 Q \geqslant 2{,}67\ \gamma \quad (5.110\text{ b})$$

Atunci obţinem corespunzător celei mai apropiate puteri a lui 2 :

$$Q \geqslant \begin{cases} 32 \text{ dacă } \gamma = 2 \\ 64 \text{ dacă } \gamma = 4 \\ 256 \text{ dacă } \gamma = 10 \\ 512 \text{ dacă } \gamma = 20 \end{cases} \quad (5.110\text{ c})$$

Acest rezultat, prezentat în fig. 5.21 este independent de dimensionalitatea masivului de procesoare, şi stabileşte că este improbabil să fie mai avantajos, datorită întîrzierilor provocate de transferuri, să se construiască masive cu o dimensiune liniară mai mare decît cîteva sute, funcţie de valoarea lui γ. Într-adevăr, cu cît sînt mai complexe procesoarele şi deci

mai mică valoarea lui γ, cu atît trebuie menținute mai mici masivele, pentru a preveni ca transferurile să domine timpul de execuție. Și pentru calculatoarele organizate binar, ca ICL DAP, γ va avea valori mici, dacă sînt folosite în calcule aritmetice cu întregi reprezentați de cuvinte scurte (să spunem cuvinte de 8 biți cînd $\gamma \approx 2$), cum este cazul în prelucrarea imaginilor (vezi § 5.5.6).

Rezultatul anterior este important deoarece apariția tehnologiei de integrare pe scară foarte mare cu 10000, sau mai multe elemente logice pe cip, face posibilă producerea unor masive foarte mari. Astfel de masive sînt atractive din punctul de vedere al costului și pot furniza, la prima vedere, performanțe înalte. Totodată, acest rezultat arată că, atunci cînd folosim algoritmii actuali, trebuie să fim conștienți de limitările ce intervin cînd dimensiunile liniare devin mari. Deși am prezentat în mod specific problema transferului în cazul algoritmului transformatei Fourier, operații de transfer similare intervin în cadrul multor algoritmi paraleli — de exemplu, algoritmul paralel pentru evaluarea recurenței de ordinul întîi prezentat în § 5.2.2. De aceea, rezultatul trebuie considerat destul de general.

Pentru a ne concentra asupra aspectelor esențiale, s-a presupus în cadrul analizei anterioare că se execută o singură transformată uni-dimensională de lungime egală cu numărul procesoarelor disponibile. Pentru o analiză a întîrzierilor de transfer pentru transformate multidimensionale multiple, care nu se potrivesc în mod necesar cu dimensiunea masivului de procesoare, recomandăm lucrările lui Jesshope (1980 a, b).

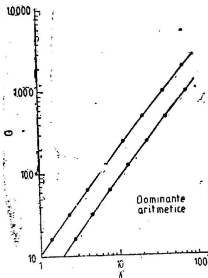


Fig. 5.21 Dimensiunea liniară Q a unui masiv de procesoare, peste care întîrzierile datorate transferurilor depășesc timpul de execuție a operațiilor aritmetice în FFT, reprezentată ca o funcție de γ, raportul dintre timpul de execuție a unei operații aritmetice și cel necesar pentru o operație de transfer. Se arată curbele corespunzătoare calculului transformatei Fourier în aritmetica complexă și transformata Rader în aritmetica întreagă modulo.

5.5.6 Transformatele teoretice numerice

Transformatele teoretice numerice, în particular transformata numerică Fermat, au aplicații importante în prelucrarea imaginilor, ca și în rezolvarea ecuațiilor cu derivate parțiale (vezi § 5.6.2), așa cum au arătat Eastwood și Jesshope (1977). Aceste transformate au fost folosite de asemenea cu folos în prelucrarea numerică a semnalelor de către Pollard (1971), Rader (1972) și Agarwal și Burrus (1974, 1975). Ele joacă un rol similar cu cel al transformatei Fourier cînd se execută operații aritmetice modulo $F_t = 2^{2^t} + 1$, al t-lea număr Fermat, iar valorile funcționale sînt limitate la întregii $0, 1, \ldots, F_t - 1$

(tehnic, un inel de întregi). În acest caz se pot defini transformatele teoretice numerice ca:

$$\hat{x}_k = \sum_{j=0}^{n-1} \alpha^{-kj} x_j \pmod{F_t} \tag{5.111 a}$$

care are inversa

$$x_j = n^{-1} \sum_{k=0}^{n-1} \alpha^{kj} \hat{x}_k \pmod{F_t} \tag{5.111 b}$$

unde α este n–a rădăcină a unității (modulo F_t). Astfel:

$$\alpha^n = 1 \tag{5.111 c}$$

Comparînd ecuațiile (5.111) cu definiția transformatei Fourier finite (5.73) și (5.74), observăm că α are aceleași proprietăți în inelul de valori întregi ca și ω în domeniul numerelor complexe. În consecință, toți algoritmii rapizi dezvoltați pentru transformata Fourier în § 5.5.2 la § 5.5.5 pot fi aplicați la fel de bine calculului transformatelor teoretice numerice, cu observația că operațiile aritmetice se execută modulo F_t. Să notăm, totuși, că de obicei se asociază factorul de scară n^{-1} la transformata inversă (5.111 b). Această convenție este inversă celei folosite la transformata Fourier, unde factorul era asociat cu transformata directă (5.71 a).

Funcțiile întregi α^{kj} și $\alpha^{-k'j}$ sînt ortogonale, a și funcțiile, ω^{kj} lucru ce poate fi demonstrat ușor:

$$\sum_{j=0}^{n-1} \alpha^{kj} \alpha^{-k'j} = \sum_{j=0}^{n-1} \alpha^{(k-k')j} \tag{5.112 a}$$

Sumînd seriile geometrice și folosind faptul că $\alpha^n = 1$, obținem:

$$\sum_{j=0}^{n-1} \alpha^{kj} \alpha^{-k'j} = \frac{\alpha^{(k-k')n} - 1}{\alpha^{k-k'} - 1} = 0, \text{ dacă } k \neq k' \tag{5.112 b}$$

Revenind la definiția (5.112 a) obținem imediat

$$\sum_{j=0}^{n-1} \alpha^{kj} \alpha^{-k'j} = \sum_{j=0}^{n-1} (\alpha^0)^2 = \sum_{j=0}^{n-1} 1 = n, \text{ dacă } k = k' \tag{5.112 c}$$

Această proprietate este analogă relației de ortogonalitate (5.72 b). Pentru cazul $t = 2$, s-au reprezentat funcțiile α^{kj} în fig. 5.22.

Tipurile cele mai folositoare de transformate numerice Fermat intervin cînd α este o putere a lui 2, cînd sînt denumite transformatele Rader (RT) (1972). În acest caz se pot executa toate înmulțirile cu puteri ale lui α din ecuația (5.111) prin deplasări, ceea ce rezultă în programe de calculator foarte rapide. Mai există o relație între numărul cifrelor binare (biților) folosiți pentru reprezentarea numerelor și lungimea n a transfor-

matei. Întregii în domeniul $0, 1, \ldots, F_t - 1 = 2^b$, unde $b = 2^t$, necesită pentru reprezentare $b + 1$ biţi, deoarece numărul $2^b = -1 \bmod F_t$ (un unu urmat de b zerouri) este inclus în domeniu. Toate celelalte numere din domeniu pot fi reprezentate cu b biţi.

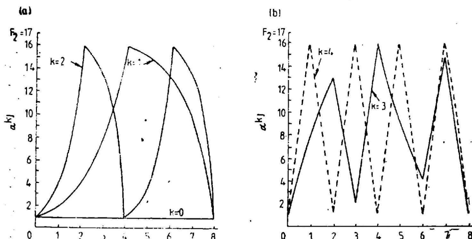


Fig. 5.22 Funcţiile ortogonale discrete α^{kj} folosite în transformata teoretică numerică Rader ($\alpha = 2$) pentru $t = 2$. Valorile funcţionale sînt în domeniul $0, 1, \ldots, F_2 - 1 = 16$ şi sînt reprezentate de 5 biţi. Lungimea transformatei este $n = 8$ ($n^{-1} = 15$), şi $k = 0, 1, \ldots, n-1 = 7$. Funcţiile pentru $k \geqslant 4$ sînt imaginea în oglindă faţă de linia verticală $j = 4$ a funcţiei desenată pentru $8 - k(\alpha^{kj} = \alpha^{(n-k)(n-j)})$.

Dacă $\alpha = 2$, lungimea transformatei este, în general

$$n = 2^b = 2^{t+1} \tag{5.113 a}$$

şi reciproca, solicitată în ecuaţia (5.111 b) este

$$n^{-1} = F_t - 2^{b-(t+1)} \tag{5.113 b}$$

Dacă se numerotează cifrele binare de la 0 la b, începînd cu bitul cel mai puţin semnificativ, n^{-1} are „1" în biţii de pe poziţiile 0 şi de la $b-1$ la $b-1-t$ inclusiv. Pentru a demonstra ecuaţia (5.113 b) se calculează $n \cdot n^{-1} = 2^{t+1}(F_t - 2^{b-(t+1)}) = 2^{t+1}F_t - 2^b$ şi, deoarece $2^b = F_t - 1$,

$$n \cdot n^{-1} = 2^{t+1}F_t - (F_t - 1)$$
$$= (2^{t+1} - 1)F_t + 1 = 1 \bmod F_t \tag{5.113 c}$$

Iată cîteva cazuri particulare de interes:

$$\alpha = 2,\ t = 3,\ b = 8, \text{biţi} = 9, n = 16, n^{-1} = 2^8 + 1 - 2^4 \tag{5.114 a}$$

$$\alpha = 2, t = 4,\ b = 16, \text{biţi} = 17, n = 32, n^{-1} = 2^{16} + 1 - 2^{11} \tag{5.114 b}$$

$$\alpha = 2, t = 5, b = 32, \text{biţi} = 33, n = 64, n^{-1} = 2^{32} + 1 - 2^{26} \tag{5.114 c}$$

$$\alpha = 2,\ t = 6,\ b = 64,\ \text{biţi} = 65,\ n = 128,\ n^{-1} = 2^{64} + 1 - 2^{57} \tag{5.114 d}$$

Transformata teoretică numerică poate fi aplicată variabilelor reale, dacă acestea sînt aduse la început în domeniul întreg permis, şi apoi discretizate la valorile întregi cele mai apropiate. În acest caz, eroarea de discretizare este de aproximativ 2^{-b}, iar dacă se doreşte o precizie mai mare se poate mări b. Apoi, dacă lungimea transformatei este prea mare, poate fi înjumătăţită prin ridicarea lui α la pătrat, atunci $\alpha = 4$, $n = b$ şi $n^{-1} = F_t - 2^{b-t}$. De exemplu,

$$\alpha = 4,\ t = 6,\ b = 64,\ \text{biţi} = 65,\ n = 64,\ n^{-1} = 2^{64} + 1 - 2^{58} \quad (5.115\ a)$$

şi prin repetarea procesului

$$\alpha = 16,\ t = 6,\ b = 64,\ \text{biţi} = 65,\ n = 32,\ n^{-1} = 2^{64} + 1 - 2^{59} \quad (5.115\ b)$$

În acest mod se poate alege o precizie b, lungimea transformatei n şi rădăcina unităţii α adecvate pentru cele mai multe circumstanţe.

Utilitatea transformatei numerice Fermat depinde în mod critic de eficienţa cu care se pot executa operaţiile aritmetice modulo F_t. Multe calculatoare execută operaţii aritmetice în complement faţă de 2 care, pentru b biţi, este modulo $2^b = F_t - 1$. Pentru a transforma aceasta la modulo F_t, trebuie scăzut 1 din suma a două numere, atunci cînd se detectează un transport de la bitul cel mai semnificativ, exceptînd cazul cînd biţii 0 la $b-1$ sînt toţi 0. Acest ultim caz reprezintă un rezultat valid şi reprezintă 2^b sau -1 mod F_t. Dacă calculatorul foloseşte numai b biţi pentru reprezentarea întregilor, se poate folosi o altă metodă, ce constă în asocierea unei variabile logice auxiliare la fiecare număr. În mod similar trebuie adunat 1 la complementul faţă de 2 negativ al unui număr, sau la diferenţa a două numere dacă rezultatul este negativ. Înmulţirea cu un număr oarecare modulo F_t se execută prin scăderea celor b biţi de rang superior ai produsului reprezentat cu 2 b biţi din biţii b de rang inferior. Observăm că $F_t = 2^b + 1$ conţine numai 2 biţi, un unu în poziţia bitului cel mai puţin semnificativ al celor b biţi de rang inferior şi un unu în poziţia bitului cel mai puţin semnificativ al celor b biţi de rang superior. În acest mod, operaţia de mai sus este aceeaşi cu normalizarea produsului la domeniul întreg permis prin scăderea lui F_t pînă ce toţi biţii de rang superior sînt 0. Trebuie din nou menţionat cazul particular cînd intervine 2^b şi pentru înmulţire cu $2^{b'}$. Înmulţirea cu puteri ale lui α se execută prin deplasarea urmată de normalizare.

Este clar că aritmetica modulo F_t necesită mai multe teste şi cazuri particulare şi este improbabil că transformatele teoretice numerice vor avea avantaje de viteză asupra transformatei Fourier pe calculatoare care execută hardware operaţiile aritmetice în virgulă mobilă. Situaţia este în mod radical diferită în cazul calculatoarelor seriale pe bit, ca **ICL DAP** şi Goodyear **STARAN**, deoarece neexistînd operatori hardware în virgulă mobilă, rutinele aritmetice speciale pot fi microprogramate la nivelul bitului pentru operaţiile aritmetice modulo. În particular, deplasările aparent necesare pentru înmulţirea cu α pot fi evitate prin adunarea biţilor corespunzători din memorie, iar orice înmulţire cu α^{kj} nu durează mai mult

decît o singură scădere necesară pentru normalizare. În tabelul 5.1 se prezintă comparaţia între operaţiile modulo F_t şi în virgulă mobilă pentru **ICL DAP**. Este clar că operaţiile modulo sînt aproximativ de 10 ori mai rapide decît cele în virgulă mobilă.

Tabelul 5.1 **Comparaţia între operaţiile modulo $F_5 = 2^{32}+1$ şi în virgulă mobilă pentru ICL DAP. Timpul în μs pentru 4096 rezultate într-un masiv de 64 × 64 procesoare**

Operaţie	Modulo F_5	Real în virgulă mobilă
+	36	150
−	30	150
×	24	250
Route	10	10

Dacă se calculează transformata Rader cu aceeaşi tehnică folosită de metoda **PARAFT** pentru transformata Fourier (vezi § 5.5.4), pentru fiecare nivel din cele $\log_2 n$ ale algoritmului se vor executa o adunare paralelă şi o înmulţire paralelă. Numărul total de operaţii aritmetice modulo F_t, fiecare cu paralelism n, pentru masivul k-dimensional de $n = Q$ procesoare din § 5.5.5. este

$$A = 2 \log_2 n = 2kq \qquad (5.116\ a)$$

Numărul de operaţii de transfer este acelaşi cu cel al algoritmului **PARAFT**, deci

$$R = k(3Q/2-2) \qquad (5.116\ b)$$

şi raportul dintre timpul consumat pentru transferuri şi cel pentru operaţii aritmetice este, pentru transformata Rader

$$t_R/t_A = \frac{3Q/2-2}{2\gamma q} \qquad (5.116\ c)$$

unde γ este raportul între timpul consumat pentru o operaţie modulo F_t şi cel pentru o operaţie de transfer unitară. Referindu-ne la tabelul 5.1 vedem că $\gamma = 3$ pentru 33 biţi şi $t = 5$ pentru ICL DAP ($k = 2$, $Q = 64$, $q = 6$), ce conduce la

$$R = 188, \quad A = 24, \quad R/A = 7{,}8 \qquad (5.117\ a)$$

şi

$$t_R/t_A = 261\% \qquad (5.117\ b)$$

Deci transferurile domină timpul de calcul aritmetic în cazul calculului transformatei Rader pe **ICL DAP** sau maşini similare.

Timpul consumat pentru transferuri va egala sau depăşi timpul consumat pentru operaţii aritmetice cînd

$$3Q/2-2 \geqslant 2\gamma q \qquad (5.118\ a)$$

sau pentru Q cînd

$$Q/\log_2 Q \geqslant 1{,}33\ \gamma \qquad (5.118\ b)$$

În fig. 5.21 se prezintă curbe pentru egalitate și regiunile planului $Q-\gamma$ unde domină transferurile sau operațiile aritmetice. Se poate vedea că pentru calculul eficient al transformatei Rader dimensiunile liniare ale masivului de procesoare trebuie menținute cît mai mici posibil. Deoarece este probabil ca γ să fie mai mic decît 4 pentru operațiile modulo, dimensiuni liniare mai mari decît 32 vor conduce la algoritmi dominați de timpul consumat pentru rearanjarea datelor în memorie mai degrabă decît de cel pentru execuția unor operații utile asupra datelor.

Analiza de mai sus s-a bazat pe o transformată egală cu dimensiunea masivului de procesoare. În cazul rezolvării unei ecuații diferențiale parțiale tri-dimensională este probabil că transformata este mult mai mare decît dimensiunea masivului în care caz influența transferurilor scade dramatic. De exemplu procentul 261% din ecuația (5.117 b) pentru o transformată 64×64, devine numai 25% pentru o transformată $64 \times 64 \times 64$ pe **ICL DAP** 64×64 (Jesshope 1980 a).

Cu toate că transferurile pot domina calculele aritmetice, performanța de ansamblu a transformatei Rader depășește cu mult pe cea a transformatei Fourier pe mașini ca **ICL DAP**. Dacă comparăm necesarul de operații aritmetice pentru a transforma în valori întregi prin transformata Rader cu cel impus de transformata Fourier a n valori reale (sau n/2 valori complexe), găsim 2 $\log_2 n$ operații paralele în comparație cu $4\log_2 n$. În plus am văzut în tabelul 5.1 că fiecare operație modulo folosită de transformata Rader este de aproximativ 10 ori mai rapidă decît aceeași operație în virgulă mobilă necesară pentru transformata Fourier. Se poate anticipa, pe această bază, un avantaj de viteză de aproximativ 20 de ori pentru transformata Rader. Valorile reale vor fi mai mici decît acest maximum datorită întîrzierilor de transfer care vor fi aceleași pentru ambele transformate. De exemplu, pentru o transformată tridimensională cu $32 \times 32 \times 16$ valori executată pe **ICL DAP** 32×32, transformata Rader durează 50 ms, iar Fourier 700 ms, folosindu-se același algoritm rapid în baza 2 (Eastwood și Jesshope 1977). Raportul de 14 în favoarea transformatei Rader este consistent cu estimările anterioare.

5.6 Ecuații diferențiale parțiale

Pentru a rezolva ecuații diferențiale parțiale (PED) pe calculatoare paralele, putem folosi rezultatele obținute mai înainte în acest capitol la rezolvarea sistemelor tridiagonale (vezi § 5.4) și la calculul transformatelor (vezi § 5.5.). Aproape toate tehnicile numerice de rezolvare a PED, care s-au folosit începînd cu 1950 pe calculatoare seriale, implică rezolvarea repetată a unor sisteme tridiagonale sau transformarea repetată a unor mulțimi independente de date. Astfel de metode sînt potrivite pentru implementarea eficientă pe calculatoare paralele deoarece sistemele sau transformatele independente pot fi calculate în paralel. Astfel, problema nu constă în introducerea paralelismului într-o metodă secvențială cunoscută, așa cum a fost cazul în secțiunile anterioare ale capitolului, ci mai degrabă în potrivirea paralelismului natural al calculatorului la paralelismul existent al algoritmului. De exemplu, algoritmul pentru rezolvarea unei pro-

bleme tri-dimensionale pe o structură $n \times n \times n$ poate fi exprimată în termenii unor operații cu paralelism n, n^2 sau n^3. Alegerea tehnicii celei mai adecvate pentru un anumit calculator implică identificarea căruia dintre aceste trei niveluri îi corespunde cel mai bine paralelismul natural al calculatorului.

Întîi (§ 5.6.1) descriem metodele iterative obișnuite (sau metodele cu relaxare) în două dimensiuni deoarece ele se pot aplica la rezolvarea celei mai generale PED liniare cu coeficienți oarecari. Facem observația importantă că metoda ajustării simultane a tuturor punctelor (metoda Jacobi) care are paralelismul maxim n^2 nu poate fi utilizată datorită convergenței extrem de lente. Acesta este un avertisment că în căutarea metodelor cu paralelism maxim, nu trebuie ignorate rezultatele de mult timp cunoscute ale analizei numerice asupra convergenței, așa cum apar de exemplu în lucrările lui Varga (1962) sau Forsythe și Wasow (1960). Metoda suprarelaxărilor succesive cu ordonare pară/impară și accelerare Cebîșev sînt recomandate datorită proprietăților superioare de convergență, deși paralelismul este înjumătățit. Această metodă poate fi aplicată prin puncte (SOR) sau prin linii (SLOR) și este interesant că noi am găsit că prima metodă este cea mai avantajoasă pe mașini ca ICL DAP, iar ultima pe mașini precum CRAY X-MP. Totuși, Hunt (1979) a arătat că poate fi aplicat în paralel prin operarea cu linii diagonale alternative în maniera unui pipeline. Paralelismul se apropie de $n^2/2$ dacă numărul de iterații este mult mai mare decît n. Grosch (1979) a studiat aplicarea metodelor iterative multi-rețea la calculatoare paralele. O altă metodă iterativă bună pe care o discutăm este metoda implicită a alternării direcției (ADI). Alte metode sînt discutate de Vajtersic (1984), iar implementări vectoriale de Schonauer (1987).

Rezolvarea PED simple cu coeficienți constanți (de exemplu ecuația Poisson $\nabla^2\Phi = \rho$, în regiuni simple (ca pătrate sau dreptunghiuri) cu condiții la margine simple (valoare dată, unghi sau periodicitate), are aplicații importante în fizică și inginerie. Sînt disponibile, în special, metode directe rapide (ne-iterative), bazate pe FFT, pentru rezolvarea acestei clase de probleme, pe care le prezentăm în § 5.6.2. Viteza acestor metode face posibilă simularea comportării în timp a stelelor în galaxii, electronilor în dispozitivele semiconductoare și a atomilor în solide sau lichide (vezi Hockney și Eastwood 1981). Din nou, algoritmii care au fost dezvoltați la început pentru calculatoare seriale, sînt inerent paraleli și pot fi implementați eficient pe calculatoare paralele, fără modificări. Astfel de metode au fost extinse de Vajtersic (1982) la ecuația biarmonică și implementate pe calculatrorul **EGPA** (§ 1.1.8).

Primele două subcapitole sînt scrise în termenii unor probleme bi-dimensionale atît pentru o prezentare ușoară cît și pentru faptul că aceste probleme sînt adecvate caracteristicilor calculatoarelor curente, c1988 și deci puse cel mai frecvent. Viteza suplimentară asociată cu noile calculatoare paralele face să fie posibilă rezolvarea problemelor tri-dimensionale cu o rezoluție rezonabilă (să zicem cu caroiaj 64 la puterea 3). De aceea vom trata în ultima parte (§ 5.6.3) cîteva din strategiile alternative posibile pentru rezolvarea unor astfel de probleme tri-dimensionale.

5.6.1 Metode iterative : SOR, SLOR, ADI

Cea mai generală PED liniară de ordinul 2 în două variabile poate fi exprimată cu :

$$A(x, y) \frac{\partial^2\Phi}{DX^2} + B(x, y) \frac{\partial\Phi}{\partial x} + C(x, y) \frac{\partial^2\Phi}{\partial y^2} +$$

$$+ D(x, y) \frac{\partial\Phi}{\partial y} + E(x, y)\,\Phi = \rho(x, y)$$

unde coeficienţii A, B, C, D, E sînt funcţii arbitrare dependente de poziţie. Această ecuaţie conţine principalele ecuaţii ale fizicii matematice şi ingineriei (ecuaţiile Helmholtz, Poisson, Laplace, Schrodinger şi de difuzie) în sistem de coordonate cunoscute (cartezian (x, y) polar (r, θ), cilindric (r, z), sferic asimetric (r, θ) şi suprafaţa sferică (θ, Φ)). Dacă se diferenţiază ecuaţia de mai sus pe un caroiaj de puncte $n \times n$, folosind procedurile standard (vezi de exemplu, Forsythe şi Wason 1960), se obţine un sistem de ecuaţii algebrice, fiecare legînd valorile variabilelor de 5 puncte vecine de pe caroiaj.

$$a_{p,q}\Phi_{p,q-1} + b_{p,q}\Phi_{p,q+1} + c_{p,q}\Phi_{p-1,q} + d_{p,q}\Phi_{p+1,q} + e_{p,q}\Phi_{p,q} = f_{p,q} \quad (5.119\text{ b})$$

unde indicii întregi p, q = 1, ..., n etichetează punctele caroiajului după direcţiile x, respectiv y. Coeficienţii a, b, c, d, e variază de la un punct la altul şi sînt legaţi de funcţiile A, B, C, D, E şi distanţele dintre puncte într-un mod complicat, produs prin aproximarea folosită pentru diferenţe. Variabila din membrul drept $f_{p,q}$ este o combinaţie liniară a valorilor $\rho_{(x;\,y)}$ în punctul (p, q). În cazul cel mai simplu reprezintă valoarea lui $\rho_{(x;\,y)}$ în punctul (p, q).

Procedurile iterative sînt lansate cu valorile reprezentative pentru $\Phi_{p,q}$ în toate punctele caroiajului, şi se foloseşte ecuaţia (5.119 b) ca bază pentru calculul valorilor îmbunătăţite. Procesul se repetă şi, dacă are succes, valorile lui Φ converg spre soluţia ecuaţiei (5.119 b) în toate punctele. În cazul procedurii celei mai simple, valorile lui Φ pentru toate punctele caroiajului sînt modificate simultan la valorile pe care le-ar avea prin ecuaţia (5.119 b) dacă se presupune că toate valorile vecine ale lui Φ sînt corecte, adică este înlocuit pentru fiecare punct de valoarea nouă :

$$\Phi^*_{p,q} = \frac{f_{p,q} - a_{p,q}\Phi_{p,q-1} - b_{p,q}\Phi_{p,q+1} - c_{p,q}\Phi_{p-1,q} - d_{p,q}\Phi_{p+1,q}}{e_{p,q}} \quad (5.120)$$

Deoarece înlocuirea trebuie să aibă loc simultan, toate valorile lui Φ sînt valori „vechi", din iteraţia anterioară, în timp ce cele din membrul stîng sînt „noile" valori, probabil îmbunătăţite.

Deoarece metoda de mai sus a înlocuirii simultane a fost propusă prima oară de către Jacobi (1845), este adeseori denumită metoda Jacobi. Ea este ideală pentru implementare pe calculatoare paralele. Modificarea simultană înseamnă că ecuaţia (5.120) poate fi evaluată în paralel pentru

toate punctele caroiajului cu paralelismul maxim posibil de n^2. De obicei $n = 32$ la 256, astfel că paralelismul variază de la aproximativ 1000 la aproximativ 64000. Se obțin astfel vectori suficient de lungi pentru o procesare eficientă pe calculatoare pipeline.. Pe de altă parte, pentru masive de procesoare cum este **ICL DAP**, se poate alege probabil caroiajul pentru a corespunde exact masivului de procesoare, sau să fie un multiplu al acestuia, facilitînd astfel utilizarea calculatorului cu toate procesoarele active (de exemplu, rezolvarea unei probleme 64×64 pe un **ICL DAP** 64×64). Mai mult, toate variabilele folosite în ecuația (5.120) aparțin unor procesoare vecine în masiv și pot fi transmise fără a folosi operații de transfer. De fapt, **ICL DAP** a fost proiectat în mod evident pentru acest tip de aplicație.

Faptul că se poate implementa eficient pe calculatoarele paralele numai o singură iterație a metodei de înlocuire simultană, nu înseamnă în mod necesar că este o metodă bună de folosit, deoarece trebuie să luăm de asemenea în considerare numărul necesar de iterații pentru a obține o convergență satisfăcătoare. Rata convergenței nu se poate afla ușor pentru ecuația generală (5.119 b) dar sînt bine cunoscute rezultate analitice pentru cazul mai simplu al ecuației Poisson în pătrat cu valori zero la margine (condițiile Dirichlet). Aceasta corespunde la considerarea valorilor $a_{p,q} = b_{p,q} = c_{p,q} = d_{p,q} = 1$ și $e_{p,q} = -4$ pentru toate punctele, ceea ce mai este cunoscut ca problema modelului. Subliniem că metodele iterative nu trebuie folosite pentru rezolvarea problemei modelului, deoarece metodele de transformare directă din § 5.6.2 sînt de cel puțin 10 ori mai rapide. Folosim aici problema modelului numai pentru a indica rata convergenței ce se poate obține pentru metodele iterative folosite la rezolvarea ecuației diferențiale generale (5.119 b), caz în care nu se pot folosi metode de transformare rapidă.

Se poate arăta (vezi de ex. Varga 1962) că, vor avea avantaje de viteză asupra transformatei Fourier pe calculatoare care execută hardware operațiile; pentru metoda Jacobi :

$$\lambda_j = \cos(\pi/n) \simeq 1 - \frac{1}{2}\pi^2/n^2 . \qquad (5.121\ a)$$

Factorul de convergență, λ_j, poate fi folosit pentru calculul lui t^*_j, numărul de iterații necesar pentru a reduce eroarea printr-un factor de 10^{-p}

$$t^*_j = \frac{-p}{\log_{10}\lambda_j} \simeq \frac{2}{\pi^2 \log_{10} e} p\, n^2 \simeq \frac{pn^2}{2} \qquad (5.121\ b)$$

Astfel, o reducere modestă a erorii de 10^{-3} pentru un caroiaj obișnuit de 128×128 ar impune un număr de aproximativ 24000 iterații. O astfel de convergență lentă face metoda Jacobi ineficientă pentru probleme practice, deși este deosebit de adecvată implementării pe calculatoare paralele.

Metoda iterativă cea mai utilizată pe calculatoare seriale este metoda suprarelaxărilor succesive prin puncte, sau SOR. În cadrul acestei metode

se foloseşte o medie ponderată a valorilor „vechi" şi stelate pentru calculul valorilor „noi"

$$\Phi^{nou}_{p,q} = \omega\Phi^{*}_{p,q} + (1-\omega)\,\Phi^{vechi}_{p,q} \tag{5.122 a}$$

unde ω este factorul de relaxare constant, de obicei în domeniul $1 \leqslant \omega \leqslant 2$, ales pentru îmbunătăţirea ratei convergenţei. Se poate arăta că, pentru problema modelului, cea mai bună rată a convergenţei se obţine cu :

$$\omega = \omega_b = \frac{2}{1+(1-\rho^2)^{1/2}} \tag{5.122 b}$$

unde $\lambda = \omega_{b-1}$, iar ρ este factorul de convergenţă a iteraţiei Jacobi corespunzătoare. De aici, $\rho = \cos(\pi/n)$ pentru problema modelului. În mod obişnuit punctele caroiajului sînt prelucrate secvenţial punct cu punct, sau linie cu linie, ca la citirea cuvintelor dintr-o carte sau tipărirea unei pagini de text. Îmbunătăţirea convergenţei se bazează pe faptul că noile valori le înlocuiesc pe cele vechi, imediat ce sînt calculate; de aici, valorile din membrul drept al ecuaţiei (5.120) folosite pentru calculul lui Φ^*, sînt o combinaţie între valorile vechi şi noi iar ecuaţia (5.120) nu poate fi calculată pentru toate punctele în paralel în modul cum se realizează cu metoda Jacobi. Ar rezulta că metoda SOR, deşi are proprietăţi superioare de convergenţă este esenţial o metodă secvenţială şi, deci, inadecvată implementării pe calculatoare paralele. Totuşi, Hunt (1979) a arătat cum poate fi folosit principiul pipelining macroscopic pentru implementarea SOR cu un paralelism maxim de $n^2/2$.

Din fericire se poate obţine o convergenţă îmbunătăţită a metodei SOR, pentru alte moduri de baleiere a caroiajului. Una din cele mai bune proceduri este ordonarea impar/par cu accelerare Cebîşev. În cadrul ei, se împrt punctele caroiajului în două grupe, depinzînd de valoarea impară sau pară a sumei $p + q$. În fig. 5.23 (a) punctele pare sînt încercuite, iar cele impare marcate cu x. Metoda constă în execuţia unor iteraţii în cursul cărora numai jumătate din puncte sînt ajustate (în mod alternativ punctele pare şi apoi impare) conform ecuaţiei (5.120) şi (5.122). În plus, la fiecare iteraţie valoarea lui ω se modifică conform :

$$\omega^{(0)} = 1$$

$$\omega^{(1/2)} = 1\Big/\left(1 - \frac{1}{2}\rho^2\right)$$

$$\omega^{(t+1/2)} = 1\Big/\left(1 - \frac{1}{4}\rho^2\omega^{(t)}\right),\ t = \frac{1}{2}, 1 \ldots, \infty \tag{5.123 a}$$

unde exponentul t marchează numărul iteraţiei. Se poate arăta că ω tinde, pentru t mare, la limita $\omega^{(\infty)} = \omega_b$, factorul de relaxare constant ce este folosit pe parcursul procedurii tradiţionale SOR (5.122 b). Factorul

de convergență asimptotică este prin urmare același pentru ambele metode de formulare a SOR și se calculează cu:

$$\lambda_{SOR} = \omega_b - 1 \simeq 1 - 2\pi/n \qquad (5.123\ b)$$

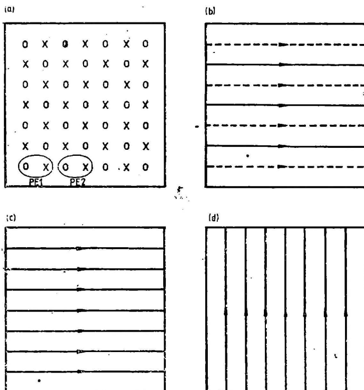


Fig. 5.23 Modelul de structurare a datelor în cadrul diverselor metode iterative: a) suprarelaxarea succesivă a punctelor (SOR); b) suprarelaxarea succesivă a liniilor (SLOR); c) și d) metoda de alternare implicită a direcției (ADI). Metoda transformatei Fourier directă multiplă (MFT) folosește aceeași structură cu ADI. Liniile al căror sens este marcat cu săgeată corespund fie unei transformate Fourier, fie unui sistem tridiagonal de ecuații.

Numărul de iterații pentru o reducere a erorii de 10^{-p} este

$$t^*_{SOR} \simeq \frac{np}{2\bar{n}\log_{10} e} \simeq \frac{np}{3} \qquad (5.123\ c)$$

Concluzionăm, pe baza ecuației (5.123 c) că o reducere a erorii de 10^{-3} pentru un caroiaj 128×128 ar necesita aproximativ 128 iterații, în comparație cu 24000 pentru metoda Jacobi. Importanța folosirii unei metode iterative cu o convergență bună este evidentă. Se poate arăta de asemenea (vezi Varga 1962) că norma maximă a vectorului eroare descrește în cazul metodei SOR cu accelerație Cebîșev și ω variabil, în timp ce în etapele inițiale ale metodei SOR tradiționale cu ω constant poate

şi, în mod frecvent, creşte (vezi Hockney 1970, fig. 7 şi 9). Deoarece metoda SOR Cebîşev nu necesită operaţii aritmetice suplimentare şi are proprietăţi mai avantajoase de descreştere a erorii iniţiale, nu pare a exista nici un motiv pentru a nu o utiliza întotdeauna.

Metoda SOR Cebîşev are avantaje suplimentare în ce priveşte implementarea pe calculatoarele paralele. Observăm din ecuaţia (5.120), că valorile marcate cu steluţă pentru un punct impar al reţelei depind numai de valorile vechi ale punctelor vecine pare, calculate în cursul ultimei iteraţii-jumătate. Astfel, toate punctele pare pot fi modificate în paralel, cu paralelism $n^2/2$ în cursul unei iteraţii-jumătate si, similar, toate punctele impare în cursul următoarei iteraţii. Timpul necesar unei iteraţii complete pentru un calculator caracterizat de un $n_{1/2}$ este proporţional cu

$$t_1 = 24 \ (n_{1/2} + n^2/2) \qquad (5.124 \text{ a})$$

deoarece ecuaţiile (5.120) şi (5.122 a) impun execuţia a 12 operaţii aritmetice pentru un punct (observăm că $1-\omega$ este precalculat şi memorat ca o singură constantă). Metoda SOR poate fi implementată şi cu un paralelism de $n/2$ prin definirea unui vector format din cele $n/2$ valori pare (sau impare) ale unei linii a caroiajului, şi prelucrînd secvenţial cele n linii. Timpul necesar execuţiei unei iteraţii complete este proporţional cu

$$t_2 = 24 \, n \, (n_{1/2} + n/2) \qquad (5.124 \text{ b})$$

O altă strategie pentru suprarelaxarea Cebîşev este ilustrată în fig. 5.23 (b). Punctele formează mulţimi pare şi impare conform liniilor la care aparţin, aşa cum se indică cu linii continui sau întrerupte. Ecuaţia de modificare (5.120) se scrie pentru toate punctele unei linii, presupunînd că toate punctele din linia de deasupra şi cea de dedesupt sînt corecte. Atunci, se calculează valorile marcate cu steluţă pentru o linie la un moment dat, cu $p = 1, 2, \ldots, n$:

$$c_{p,q}\Phi^*_{p-1,q} + e_{p,q}\Phi^*_{p,q} + d_{p,q}\Phi^*_{p+1,q} = f_{p,q} - a_{p,q}\Phi_{p,q-1} - b_{p,q}\Phi_{p,q+1} \qquad (5.125)$$

Ecuaţia (5.125) este un sistem tridiagonal pentru toate valorile marcate cu steluţă de-a lungul unei linii, cu membrul drept depinzînd de valorile cunoscute ale liniei de deasupra şi de dedesubt. Metoda SLOR cu accelerare Cebîşev foloseşte ecuaţiile (5.22) şi (5.123 a), lucrează în mod linie, iar acum, factorul de convergenţă al metodei Jacobi este, pentru problema modelului:

$$\rho = \cos(\pi/n)/[2-\cos(\pi/n)] \simeq 1-\pi^2/n^2 \qquad (5.126 \text{ a})$$

Ecuaţia (5.122 b) este valabilă, cu noua valoare a lui ρ, ceea ce conduce la

$$\lambda_{SLOR} = \omega_b - 1 \simeq 1 - \sqrt{2}(2\pi/n) \qquad (5.126 \text{ b})$$

şi

$$t^*_{SLOR} = t^*_{SOR}/\sqrt{2} \simeq np/4 \qquad (5.126 \text{ c})$$

Astfel, asimptotic, numărul de iterații necesar pentru o reducere dată a erorii este în cazul metodei SLOR de $2^{-1/2} = 0,7$ din numărul iterațiilor necesare pentru metoda SOR.

Deci, metoda SLOR constă din rezolvarea a n/2 sisteme tridiagonale de lungime n la fiecare iterație jumătate. Dacă se rezolvă sistemele în paralel cu cel mai bun algoritm serial (procedura MULTGE din § 5.4.4 cu m = n/2) obținem un paralelism de n/2 și un timp necesar pentru execuția unei iterații complete proporțional cu

$$t_3 = 24\, n(n_{1/2} + n/2) \tag{5.127 a}$$

Pentru a obține această estimare am considerat 4 operații aritmetice pentru fiecare punct, pentru a forma membrul drept al ecuației (5.125), în plus la cele 8 operații necesare pentru fiecare punct în cazul rezolvării sistemului tridiagonal general. Se introduce un factor de 2, deoarece fiecare iterație este construită din două iterații jumătate. În alt mod, sistemele pot fi rezolvate cu paralelism $n^2/2$ cu algoritmul **PARACR**$_{par}$ (vezi § 5.4.4). Timpul necesar execuției unei iterații este în acest caz proporțional cu

$$t_4 = 2(n_{1/2} + n^2/2)\,(4 + 12\log_2 n) \tag{5.127 b}$$

unde numărul 4 din ultimul factor se referă la calcularea părților din membrul drept, iar termenul logaritmic este pentru rezolvarea sistemului tridiagonal cu algoritmul **PARACR** (vezi ecuația (5.66 b).).

Am văzut că ambele metode, SOR și SLOR, pot fi implementate cu paralelism fie de n/2, fie de $n^2/2$. Care implementare este cea mai bună, va depinde în primul rînd de apropierea valorilor n/2 sau $n^2/2$ de paralelismul hardware natural al calculatorului. Vom considera cele două cazuri extreme ale calculatoarelor **ICL DAP** cu paralelismul natural de 64×64 și **CRAY X-MP** cu paralelismul natural 64. Rețele ce intervin în mod real, cu o rezoluție bună au probabil între 32 și 256 puncte pentru fiecare direcție. Astfel este natural de a exploata paralelismul calculatorului **CRAY X-MP** cu o parte a rețelei, pentru care să se folosească algoritmi ce au paralelism n sau n/2. Pe de altă parte, este natural de a folosi paralielismul calculatorului **ICL DAP** pentru algoritmi cu paralelism n^2 sau $n^2/2$. Această alegere este și mai convingătoare dacă ne gîndim că **ICL DAP** este un masiv bidimensional de procesoare.

După alegerea nivelului paralelismului care este cel mai indicat pentru calculatorul respectiv, trebuie văzut care este cel mai bun algoritm, Cebîșev SOR sau Cebîșev SLOR. La **CRAY X-MP** cu paralelism n/2, SOR va fi algoritmul cel mai bun dacă

$$t_2 < t_3/\sqrt{2} \tag{5.128 a}$$

unde $\sqrt{2}$ reflectă convergența mai bună a metodei SLOR. Deoarece $t_2 = t_3$ [vezi ecuațiile (5.124 b) și (5.127 a)], condiția de mai sus (5.128 a) nu poate fi respectată niciodată, deci SLOR va fi întotdeauna cel mai bun algoritm pentru CRAY X-MP. Formulat altfel, timpul necesar unei iterații este același pentru paralelism n/2 și SLOR cîștigă datorită convergenței ei mai rapide.

Pentru **ICL DAP** cu paralelism $n^2/2$, SOR va fi algoritmul cel mai bun dacă

$$t_1 < t_4/\sqrt{2} \qquad (5.128\ b)$$

sau

$$24 < \sqrt{2}\,(4 + 12 \log 2n) \qquad (5.128\ c)$$

Această condiţie este satisfăcută pentru $n \geqslant 2$, deci pentru toate reţelele. De aceea, pentru **ICL DAP** implementarea metodei SOR este cea mai avantajoasă. Rezultatul este determinat de faptul că rezolvarea paralelă a ecuaţiilor tridiagonale prin reducere ciclică necesară la rezolvarea SLOR introduce operaţii aritmetice suplimentare. În acest caz, convergenţa mai bună a metodei SLOR nu este suficientă.

Ultima metodă iterativă ce poate fi folosită este metoda de modificare implicită a direcţiei (ADI). Aceasta este reprezentată în fig. 5.23 (c şi d). Cînd se foloseşte la rezolvarea ecuaţiilor diferenţiale

$$(L_x + L_y)\,\Phi = \rho \qquad (5.129\ a)$$

unde L_x şi L_y sînt matrici ce reprezintă operatorii cu diferenţe finite după direcţiile x şi y, soluţia se determină iterativ, plecînd de la o valoare iniţială cu

$$(I + r_n L_x)\,\Phi^{(n+1/2)} = (I - r_n L_y)\,\Phi^{(n)} + r_n q \qquad (5.129\ b)$$

$$(I + r_n L_y)\,\Phi^{(n+1)} = (I - r_n L_x)\,\Phi^{(n+1/2)} + r_n q \qquad (5.129\ c)$$

unde $n = 0, 1, \ldots$ este numărul iteraţiei, iar parametrul r_n se modifică la fiecare baleiere dublă în sensul îmbunătăţirii convergenţei aproximărilor $\Phi^{(n)}$ ale soluţiei Φ. Dacă ne interesează PDE generală liniară de ordinul doi (ecuaţia (5.119 a)), L_x şi L_y sînt matrici tridiagonale. Ecuaţia (5.129 b) implică calculul membrului drept la costul a 7 operaţii pentru fiecare punct, urmate de rezolvarea a n sisteme tridiagonale independente (unul pentru fiecare linie orizontală a caroiajului) de lungime n fiecare (vezi fig. 5.23(c).). Evident, aceste sisteme pot fi rezolvate în paralel fie cu paralelism n, fie n^2, ca în cazul metodei SLOR. Iteraţia se execută cu un proces de calcul similar, doar că acum sistemele tridiagonale se rezolvă de-a lungul fiecărei linii verticale a caroiajului (vezi fig. 5.23(d)).

Dacă paralelismul utilizat este n (de exemplu, pentru **CRAY X-MP**), folosim metoda **MULTGE**, iar timpul necesar execuţiei unei iteraţii este proporţional cu

$$t_5 = 30n\,(n_{1/2} + n) \qquad (5.130\ a)$$

Pe de altă parte, folosind un paralelism n^2 (ICL DAP, de exemplu) şi metoda $PARACR_{par}$, timpul de calcul este proporţional cu

$$t_6 = 2(n_{1/2} + n^2)\,(7 + 12\log_2 n). \qquad (5.130\ b)$$

Faptul că datele sînt accesate întîi după linii orizontale şi apoi după linii verticale poate complica implementarea ADI pe unele calculatoare. Dacă caroiajul este astfel memorat încît elementele adiacente în liniile verticale sînt adiacente şi în memoria calculatorului (modul de memorare **FORTRAN**, după coloane), atunci rezolvarea ecuaţiei (5.129 b) nu va pune probleme deoarece incrementul între elementele vectorului este unitatea. Totuşi, în rezolvarea ecuaţiei (5.129 c) incrementul dintre elementele vectorului va fi egal cu numărul de variabile dintr-o coloană. Dacă este o putere a lui 2, conflictele de acces la memorie pot fi o problemă chiar şi pentru calculatoare ca **CRAY X-MP**, care permit utilizarea incremenţilor diferiţi de unitate. Problema poate fi evitată prin memorarea structurii ca şi cum ar avea lungimea coloanei mai mare cu 1 faţă de cea reală. Pe de altă parte, pe calculatoare ca **CDC CYBER 205** care acceptă numai vectori cu increment unitate, cel de-al doilea pas al metodei ADI poate fi executat numai după o rotaţie a întregii structuri în memorie. Costul acestei operaţii poate face ADI o metodă neatractivă pentru astfel de maşini, în comparaţie cu metoda SLOR care accesează datele numai în direcţie orizontală. Experienţa aplicării metodelor ADI şi SLOR este variată şi nu există nici un mod a priori de a determina care este metoda cea mai avantajoasă pentru o anumită problemă. Poate fi necesar să se încerce ambii algoritmi.

5.6.2 Metode directe : MFT, FACR (l)

Metodele directe sînt acelea care permit obţinerea soluţiei într-un număr finit de paşi sau operaţii aritmetice. Deoarece nu se folosesc iteraţii, eficienţa lor nu depinde de calitatea valorilor iniţiale sau de abilitatea de a judeca corect cînd să fie oprit procesul iterativ. Pe de altă parte, metodele directe de rezolvare a ecuaţiei liniară generală cu diferenţe finite (5.119 b) se pot aplica numai caroiajelor cu pînă la 1000 puncte, care nu sînt un obiectiv prea important pentru calculul paralel. De aceea, vom discuta tehnicile de transformare speciale care sînt disponibile pentru rezolvarea ecuaţiei Poisson în pătrat sau dreptunghi, cu condiţiile la graniţă

$$\Phi_{p,q-1} + \Phi_{p,q+1} + \Phi_{p-1,q} + \Phi_{p+1,q} - 4\Phi_{p,q} = f_{p,q}$$

$$p, q = 0, 1, \ldots, n-1 \qquad (5.131)$$

unde coeficienţii sînt constanţi : $a_{p,q} = b_{p,q} = c_{p,q} = d_{p,q} = 1$ şi $e_{p,q} = -4$. Aceste metode, denumite adeseori rezolvitori rapizi eliptici sau algoritmi RES, sînt mai rapide cu un ordin de mărime decît metodele iterative descrise în § 5.6.1 şi pot fi aplicate caroiajelor cu 10 000 sau mai multe puncte. Metodele RES sînt inerent înalt paralele şi deci foarte adecvate pentru implementarea pe calculatoare paralele. Rezolvitorii eliptici rapizi au fost analizaţi de Swarztrauber (1977) şi de Hockney (1980), iar performanţele diverşilor algoritmi, comparate de Hockney (1970, 1978) şi Temperton (1979 a) pentru cazul calculatoarelor seriale şi de Grosch (1979) şi Temperton (1979 b, 1980) pentru calculatoare paralele.

Rezolvitorii eliptici rapizi sînt definiţi ca fiind acele metode cu un număr de operaţii de ordinul $n\log_2 n$ sau mai bine [de exemplu, FACR(l) sînt de ordinul $n \log_2(\log_2 n)$], care sugerează imediat uitilizarea algoritmilor pentru transformate rapide (vezi § 5.5). Cea mai simplă metodă se obţine prin considerarea unei transformate Fourier dublă a ecuaţiei (5.131)

$$[2 \cos (2\pi k/n)+2 \cos (2\pi l/n)-4]\ \bar{\Phi}^{k,l} = \bar{f}^{k,l} \qquad (5.132\ a)$$

unde

$$\bar{\Phi}^{k,l} = \frac{1}{n^2} \sum_{p,q=0}^{n-1} \Phi_{p,q} \exp[-2\pi i\,(kp + lq)/n] \qquad (5.132\ b)$$

şi

$$\Phi_{p,q} = \sum_{k,l=0}^{n-1} \bar{\Phi}^{p,q} \exp[+2\pi i\ (kp + lq/n] \qquad (5.132\ c)$$

şi în mod similar pentru $\bar{f}^{k,l}$ şi $f_{p,q}$. Ecuaţia (5.132 a) permite calculul amplitudinilor armonice Fourier a soluţiei, $\bar{\Phi}^{k,l}$, prin împărţirea armonicelor din membrul drept, $\bar{f}^{k,l}$, cu factorii numerici cunoscuţi din parantezele pătrate. Următoarea metodă de transformare Fourier multiplă (MFT) se impune de la sine:

a) Analiza Fourier $f_{p,q}$, folosind FFT, $f_{p,q} \rightarrow \bar{f}^{k,l}$;

b) Împărţirea paralelă a ecuaţiei (5.132 a) cu $[\ldots]$, $\bar{f}^{k,l} \rightarrow \bar{\Phi}^{kl}$;

c) Sinteza Fourier $\bar{\Phi}^{k,}$ folosind FFT, $\Phi^{k,l} \rightarrow \Phi_{p,q}$.

Atît analiza cît şi sinteza Fourier pot fi executate convenabil prin transformarea la început a tuturor liniilor de date după direcţia x, ca în fig. 5.23(c) şi apoi transformarea datelor rezultate pe linie în direcţia y, ca în fig. 5.23(d). Aceasta se poate interpreta în mod evident ca reformularea echivalentă expresiei (5.123 b)

$$\Phi^{k,l} = \frac{1}{n} \sum_{q=0}^{n-1} \left(\frac{1}{n} \sum_{p=0}^{n-1} \Phi_{p,q} \exp(-2\pi i k p/n \right) \exp\,(-2\pi i l q/n) \qquad (5.134)$$

unde suma interioară este transformată după x, iar cea exterioară transformată după y. Am considerat, mai sus, condiţii la graniţă cu periodicitate dublă. Condiţiile Dirichlet (valoare dată) pot fi atinse dacă se foloseşte transformata finită sine, iar condiţiile Neumann (gradient dat), dacă se foloseşte transformata finită cosine. Aceste rafinamente nu modifică cele spuse privind implementarea paralelă.

Transformatele după linie din ecuaţia (5.134) sînt toate independente şi pot fi executate în paralel cu paralelism n sau n^2. În primul caz, se execută în paralel n transformate reale cu cel mai bun algoritm serial, adică metoda **MULTFT** definită de ecuaţia (5.100 a) cu $m = n$. Timpul total de execuţie al algoritmului este

$$t_{MULTFT} = 4 \times \frac{1}{2} \times 5n\,(n_{1/2} + n)\log_2 n \qquad (5.135\ a)$$

$$= 10\, n\, (n_{1/2} + n) \log_2 n \qquad (5.135\ b)$$

Factorul 4 din ecuația (5.135 a) vine de la necesitatea de a transforma toate punctele de 4 ori — analiza și sinteza în direcțiile x și —y, iar factorul $\frac{1}{2}$ datorită execuției transformatelor reale, și nu copmlexe. Dacă se execută n transformate în paralel cu algoritmul $\mathbf{PARAFT_{par}}$, paralelismul este n^2 și

$$t_{PARAFT_{par}} = 4 \times \frac{1}{2} \times 8(n_{1/2} + n^2)\log_2 n \qquad (5.136\ a)$$

$$= 16\,(n_{1/2} + n^2)\log_2 n \qquad (5.136\ b)$$

Alegerea paralelismului va urma același raționament cu cel de la metodele iterative. Probabil vom folosi paralelismul n pe calculatorul **CRAY X-MP** și n^2 pe calculatoare ca **ICL DAP**. Deoarece ambele nivele de paralelism sînt la fel de adecvate pentru hardul calculatoarelor, ne putem întreba care algoritm este cel mai bun. Algoritmul $\mathbf{PARAFT_{par}}$ va avea o performanță mai mare dacă

$$t_{MULTFT} > t_{PARAFT} \qquad (5.137\ a)$$

de unde

$$\frac{n_{1/2}}{n} > \frac{6n}{10n - 16} \qquad (5.137\ b)$$

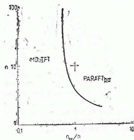


Fig. 5.24 Regiunile din planul $(n_{1/2}/n, n)$ ce favorizează implementarea metodei MFT, fie cu algoritmul MULTFT, fie cu $PARAFT_{par}$. Conform ecuației (5.137b).

În fig. 5.24 se prezintă zonele din planul $(n_{1/2}/n)$, n) unde cei doi algoritmi sînt mai buni, conform ecuației (5.137 b). Observăm că, pentru vectori cu o lungime rezonabilă (să spunem $n > 10$), metoda $PARAFT_{par}$ este cea mai avantajoasă dacă $(n_{1/2}/n)$, $\gtrsim 0{,}5$. Cu alte cuvinte, nu este surprinzător că dacă calculatorul „apare" paralel cînd este evaluat cu scala vectorului $(n_{1/2} \gtrsim n/2)$, algoritmul proiectat pentru a maximiza numărul de operații paralele este cel mai avantajos.

Transformata Fourier multiplă mai este folosită la rezolvarea unor probleme înrudite exprimate cu convoluții. De exemplu, potențialul unui punct de pe caroiaj poate fi scris ca suma contribuțiilor tuturor celorlalte puncte ca surse, multiplicată cu un factor cunoscut (funcția G a lui Green), care depinde numai de separarea între sursă și punct. Astfel,

$$\Phi_{p,q} = \sum_{s=0}^{n-1}\sum_{t=0}^{n-1} G_{p-s,q-t} f_{s,t} \qquad (5.138)$$

Acest mod de formulare a problemei potențialului este adecvat cînd sursele formează un grup izolat și singura condiție este că potențialul scade constant spre ∞. Un exemplu este calculul potențialului unui grup de galaxii sau stele (vezi Hockney 1970), în care caz G reprezintă interacțiunea columbică r^{-1}, unde r este distanța dintre sursă și punctul de potențial analizat. Expresia anterioară mai este folosită în metoda P^3M (Hockney și Eastwood 1988) pentru calculul potențialului în sistemele mari de ioni ce interacționează.

Ecuația (5.138) este echivalentă cu asertiunea că potențialul este convoluția funcției de distribuție a sursei f cu funcția Green, G. Teorema convoluției (vezi Bracewell 1965) stabilește că transformata Fourier a convoluției este proporțională cu produsul transformatelor Fourier ale funcțiilor implicate. Considerînd transformata ecuației (5.138) obținem :

$$\bar{\Phi}^{k,l} = n^2 \bar{G}^{k,l} \bar{f}^{k,l}, \quad k, l = 0, 1, \ldots, n-1 \qquad (5.139\ a)$$

Procedeul de rezolvare decurge astfel

$$f_{p,q} \xrightarrow{FFT} \bar{f}^{k,l} \xrightarrow{\times \bar{G}^{k,l}} \Phi^{k,l} \xrightarrow{FFT} \Phi_{p,q} \qquad (5.139\ b)$$

Este aceeași procedură cu (5.133) unde înmulțirea cu G înlocuiește împărțirea la [ ] a ecuației (5.132 a), iar algoritmul **MULTFT** sau **PARAFT** pot fi folosite, conform celor discutate anterior.

Metoda de transformare schițată în paragraful anterior se bazează pe existența teoremei de convoluție pentru înlocuirea sumei multiple (5.138) cu înmulțirea simplă paralelă din ecuația (5.139 a). Plecînd de la ecuația (5.111) se poate arăta ușor că transformata teoretică numerică Fermat (NTT) respectă de asemenea o teoremă de convoluție. Deci, metoda definită de ecuația (5.139 b) poate fi folosită cu FFT în loc de NTT, iar operațiile în virgulă mobilă înlocuite cu operații întregi modulo, conform § 5.5.6. Eastwood și Jesshope (1977) prezintă în detaliu această metodă, recomandînd-o pentru rezolvarea anumitor PDE tridimensionale pe calculatoare ca **ICL DAP**. James și Parkinson (1980) au stabilit eficiența metodei pentru rezolvarea ecuației Poisson în cazul unei probleme gravitaționale pe o structură $65 \times 65 \times 665$.

Unul din primii și cei mai apreciați rezolvitori eliptici rapizi este metoda de analiză Fourier cu reducere ciclică a algoritmului FACR (Hockney 1965, 1970, Temperton 1980). Aceasta a fost folosită pentru minimizarea numărului de operații aritmetice pe calculatoare seriale, prin reducerea dimensiunii transformatei Fourier, și are un număr de operați asimptotic egal cu $4{,}5\ n^2 \log_4 (\log_2 n)$, pentru o utilizare optimală (vezi Hockney și Eastwood 1988). Algoritmul FACR este eficient și pe calculatoare paralele.

Ecuația Poisson discutată (5.131) poate fi rescrisă :

$$\Phi_{q-1} + A\Phi_q + \Phi_{q+1} = f_q, \; q = 0, 1, \ldots, n-1 \qquad (5.140)$$

unde elementele vectorilor coloana Φ_q şi f_q sînt, respectiv, valorile de potenţial şi membrul drept după linia orizontală q a caroiajului. Matricea tridiagonală $\mathbf{A}$, $n \times n$, are o diagonală -4 şi diagonalele de deasupra şi de dedesupt unitate. Această matrice reprezintă diferenţierea ecuaţiei după direcţia x.

Dacă înmulţim fiecare ecuaţie de linie pară, ca ecuaţia (5.140) cu $-\mathbf{A}$ şi adunăm ecuaţiile corespunzătoare liniilor impare de deasupra şi dedesupt, obţinem o mulţime de ecuaţii ce leagă numai liniile pare

$$\Phi_{q-2} + \mathbf{A}^{(1)}\Phi_q + \Phi_{q+2} = f_q^{(1)} \tag{5.141a}$$

unde

$$\mathbf{A}^{(1)} = 2\mathbf{I} - \mathbf{A}^2 = (\sqrt{2}\,\mathbf{I} + \mathbf{A})(\sqrt{2}\,\mathbf{I} - \mathbf{A}) \tag{5.141 b}$$

şi

$$f_q^{(1)} = f_{q-1} - \mathbf{A}f_q + f_{q+1} \tag{5.141 c}$$

Aceasta reprezintă un nivel al reducerii ciclice, pe o linie, a ecuaţiilor iniţiale. Ecuaţiile rezultate 4.141 a) sînt jumătate ca număr (numai linii pare) şi au aceeaşi formă cu cele iniţiale (5.140). Procesul de reducere ciclică poate fi continuat recursiv, producînd mulţimi reduse de ecuaţii pentru fiecare a patra, a opta, a şaisprezecea linie, etc., la nivelele $r = 2, 3, 4$ etc. La fiecare nivel există o nouă matrice centrală $\mathbf{A}$ (r) care poate fi exprimată ca produs a $2^{(r)}$ matrici tridiagonale

$$\mathbf{A}^{(r)} = -\prod_{k=1}^{2r} (\mathbf{A} - \beta_k \mathbf{I}),\ \beta_k = 2\cos\left(\frac{(2k-1)\pi}{2^{\pi+1}}\right) \tag{5.142}$$

Factorizarea pentru $r = 1$ este prezentă în expresia (5.141 b) a ecuaţiei.

Dacă reducerea ciclică este oprită la nivelul $r = 1$, ecuaţiile obţinute sînt

$$\Phi_{q-2}l + \mathbf{A}^{(1)}\Phi_q + \Phi_{q+2}{}^{l} = f_q^{(l)},\ q = (0, 1, \ldots) \times 2^l \tag{5.143 a}$$

Aceste ecuaţii sînt rezolvate prin analiza Fourier după direcţia x, de-a lungul vectorilor, conducînd la ecuaţii armonice independente pentru fiecare armonică k

$$\bar{\Phi}_{q-2}^k{}^l + \lambda_k \bar{\Phi}_q^k + \bar{\Phi}_{q+2}^k{}^l = \bar{f}_q^k \quad k = 0, 1, \ldots, n-1 \tag{5.143 b}$$

unde λ_k este o constantă pentru fiecare armonică, ce depinde de condiţiile la graniţă în x şi rădăcinile ale ecuaţiei (5.142). Ecuaţiile (5.143 b) sînt sisteme n tridiagonale de lungime $n2^{-l}$ şi se rezolvă cu uşurinţă. Procedura de rezolvare a ecuaţiei (5.143 a) este

$$f_q \xrightarrow[\text{analiză}]{\text{FFT}} = \bar{f}_q^{(k)} \xrightarrow[\text{ec. (5.143 b)}]{\text{rezolvarea}} \bar{\Phi}_q^k \xrightarrow[\text{sinteză}]{\text{FFT}} \Phi_q \tag{5.143 c}$$

Găsind soluţia fiecărei linii 2^l din ecuaţia 5.143 c), liniile intermediare pot fi obţinute succesiv din ecuaţiile de nivel intermediare

$$\mathbf{A}^{(r)}\Phi_q = f_q^{(r)} - \Phi_{q-2^r} - \Phi_{q+2^r} \quad \text{pentru} \quad r = l-1, l-2, \ldots, 0 \qquad (5.144)$$

Se găseşte, la aplicarea a $n2^{-(r+1)}$ linii necunoscute, că valorile lui din membrul drept al ecuaţiei (5.144) sînt valori calculate la nivelele anterioare mai adînci. Datorită factorizării (5.142), rezolvarea ecuaţiei (5.144) necesită rezolvarea succesivă a 2^r sisteme tridiagonale.

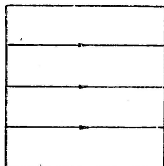
Algoritmul de mai sus este denumit **FRAC** (l) unde argumentul reprezintă numărul de nivele ale reducerii ciclice executate înainte ca ecuaţia să se rezolve ca analiză Fourier. Hockney (1978) a arătat numeric, iar Swarztrauber (1977) a demonstrat analitic că există o valoare optimă $l = l^* \approx \log_2(\log_2 n)$ care conduce la numărul minim de operaţii aritmetice. Cu cît creşte numărul reducerilor ciclice (prin creşterea lui l), cu atît se execută mai puţine transformate Fourier (sînt mai puţine linii); totuşi, trebuie rezolvate mai multe sisteme tridiagonale cu ecuaţia (5.144) pentru a obţine liniile intermediare. De aceea, valoarea exactă a optimului l^* depinde de eficienţa relativă a codului folosit pentru FFT şi rezolvarea sistemelor tridiagonale. Un cod FFT mai bun va conduce la un l^* mai mic, iar un rezolvitor tridiagonal mai bun la l^* mai mare. Probabil cea mai bună strategie este de a scrie un cod pentru un l generic, şi să se măsoare valoarea optimă. Temperton (1980) a realizat aceasta pe **IBM** 360/195 cu codul denumit PSOLVE (optim $l^* = 2$), şi de Temperton (1979 b) pe **CRAY**-1 (optim $l^* = 2$ în modul scalar, $l^* = 1$ în modul vectorial). Este evident că introducerea paralelismului la implementarea **FACR**(1) conduce optimul l^* la o valoare mai mică. Vom vedea că acest rezultat empiric concordă cu estimările teoretice simple ale performanţei, pe care le prezentăm în continuare.

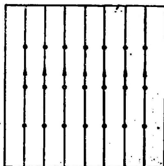
Din descrierea anterioară este evident că dacă reducerea ciclică se execută cu $l \approx \log_2 n$, va rămîne o singură ecuaţie de rezolvat pentru linia centrală. Aceasta are forma ecuaţiei (5.144) unde valorile lui Φ din membrul drept sînt, funcţie de condiţiile la graniţă, fie valorile la graniţă cunoscute, fie aceleaşi cu linia din membrul stîng. Deci, ecuaţia poate fi reprezentată fără analiza Fourier, la fel cu toate liniile intermediare. Metoda reducerii ciclice complete (**CLCR**), care poate fi numită procedură **FACR** ($\log_2 n$), a fost enunţată de Buneman (1969) care a arătat cum, la costul unor operaţii aritmetice suplimentare, procedura poate fi făcută numeric stabilă. Analiza teoretică a stabilităţii numerice a reducerii ciclice pe linie a fost formulată ulterior de Buzbee et al (1970).

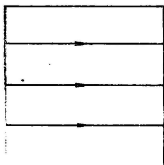
Metoda **FACR**(1) poate fi programată folosind fie forma instabilă a reducerii ciclice descrisă anterior, fie procedura stabilizată a lui Buneman, dar aceasta are un efect marginal asupra valorii optime a lui l^*. În practică, se găseşte că reducerea instabilă poate fi folosită pentru valori mici ale lui $l = 1, 2$, utilizată de **FACR**(1), iar că reducerea stabilizată trebuie folosită dacă reducerea ciclică este folosită pentru l în domeniul 5—7. Deoarece **CLCR** foloseşte valori ale lui l depărtate de optim şi necesită operaţii aritmetice suplimentare pentru stabilizare, nu o vom trata în continuare. Cu toate acestea, deşi ambele metode au fost cunoscute şi

publicate în 1970, aplicarea metodei SLCR adominat literatura de specialitate în cursul anilor '70 (vezi, de exemplu, trecerea în revistă a lui Swarztrauber 1977). Performanţa relativă a reducerii complete **CLCR** ca şi reducerea parţială caracteristică pentru **FACR**(l) au fost testate sub forma a aproximtativ 20 rezolvitori diferenţiali Poisson la Karlsruhe în 1977 (Schumann 1978). Acest concurs a fost cîştigat de programul PSOLVE al lui Temperton, un algoritm **FACR** (3) stabilizat, care a fost de 1,8 ori mai rapid decît cel mai bun program ce a folosit reducerea ciclică completă. Este interesant faptul că eroarea măsurată pentru ansamblul calculelor a fost mai mică pentru o procedură **FACR** (1) „nestabilizată", denumită POT1, decît pentru programul stabilizat. Aceste rezultate ce favorizează metoda **FACR**(l) au fost obţinute pe calculatoare seriale. Subliniem că argumentele împotriva utilizării reducerii complete sînt mai puternice cînd se ia în considerare implementarea pe calculatoare paralele, aşa cum vom vedea.

În continuare trecem în revistă paşii de execuţie ai algoritmilor FACR(l), ca şi timpul de execuţie pe un calculator paralel caracterizat de un $n_{1/2}$. În fig. 5.25 se prezintă modalităţile de structurare a datelor la







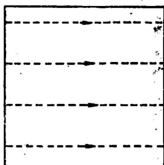


Fig. 5.25 Modelele de structurare a datelor în paşii diferiţi ai algoritmului FACB (1). Săgeţile ndică fie o transformată Fourier, fie un sistem tridiagonal de ecuaţii.

pași diferiți, pentru cazul algoritmului **FACR**(1), unde se execută o singură reducere ciclică inițială.

Algoritmul **FACR**(l) constă din 5 etape și poate fi implementat fie pentru minimizarea numărului total de operații aritmetice, s (varianta serială), fie pentru a minimiza numărul total de operații vectoriale, q (varianta paralelă), așa cum s-a specificat în § 5.1.6. Varianta serială este denumită SERIFACR, iar cea paralelă PARAFACR. Vom descrie cele două metode și le vom compara cu metoda $n_{1/2}$ de analiză a algoritmilor. Obiectul analizei este de a găsi care este varianta cea mai bună pentru un anumit calculator și, în al doilea rînd, de a alege valoarea optimă a lui l. Aceasta se va realiza prin desenarea diagramelor de fază algoritmică (Hockney 1983).

(*i*) *Algoritmul* **SERIFACR**

În această versiune a algoritmului, lungimea vectorului este proporțională cu n, numărul de puncte după o direcție a caroiajului. Cele 5 etape ale algoritmului FACR(l) sînt:

(a) *Modifică RHS* — pentru $r = 1, 2, \ldots, l$ modifică membrul drept pe $n2^{-r}$ linii în paralel folosind generalizarea ecuației (5.141 c)

$$f_q^{(r)} = f_{q-2^{r-1}} - A^{(r-1)}f_q + f_{q+2^{r-1}} \qquad (5.145\ a)$$

la costul a $(3 \times 2^{r-1}+2)\, n$ opera-ții aritmetice pe linie

$$t_a = \sum_{r=1}^{l} (n_{1/2} + n \cdot 2^{-r})(3 \times 2^{r-1} + 2)n \qquad (5.145\ b)$$

(b) *Analiza FFT* — pe $n2^{-l}$ linii în paralel folosind algoritmul MULTFT Fiecare transformare este reală, și de lungime n:

$$t_b = (n_{1/2} + n \cdot 2^{-l}) \cdot 2\frac{1}{2} \cdot n \cdot \log_2 n \qquad (5.145\ c)$$

(c) *Rezolvarea ecuațiilor armonice* — n ecuații tridiagonale, fiecare de lungime $n2^{-l}$ rezolvate în paralel cu algoritmul MULTGE

$$t_c = 5\,(n_{1/2} + n)n \cdot 2^{-l} \qquad (5.145\ d)$$

Coeficientul este 5 și nu 8 deoarece ecuația (5.143 b) are doi coeficienți care sînt unitatea ($a_i = c_i = 1$).

(d) *Sinteza FFT* — pe $n2^{-l}$ linii în paralel folosind algoritmul MULTFT. Fiecare transformată este reală și de lungime n

$$t_d = (n_{1/2} + n \cdot 2^{-l})2 \cdot \frac{1}{2} \cdot n \cdot \log_2 n \qquad (5.145\ e)$$

(e) *Obținerea liniilor intermediare* prin rezolvarea ecuației (5.144), ce implică $(2+5\times 2^r)n$ operații pe linie pentru $n2^{-(r+1)}$ linii și $r=l-1, l-2, \ldots, 0$

$$t_e = \sum_{r=0}^{l-1} (n_{1/2} + n\cdot 2^{-(r+1)})(5 \times 2^r + 2)\, n \qquad (5.145f)$$

$$= \sum_{r=1}^{l} (n_{1/2} + n\,.2^{-r})\,(5\times 2^{r-1} + 2)\, n \qquad (5.145\,g)$$

Timpul total de execuție pentru algoritmul FACR(l) este

$$t_{SERIFACR} = \sum_{r=1}^{l} (n_{1/2} + n2^{-r})\,(8\times 2^{r-1} + 4)\, n + 5\,(n_{1/2} + n\cdot 2^{-l})\, n \log_2 n$$

$$+ 5\,(n_{1/2} + n)\, n\cdot 2^{-1} \qquad (5.146\,a)$$

Evaluînd sumele obținem timpul pentru un punct

$$n^{-2}t_{SFRIFACR} = [4l + 4 + (1+5\log_2 n)2^{-l}]$$

$$+ \frac{n_{1/2}}{n}\,[4l + 8\times 2^{l} - 8 + 5\times 2^{-1} + 5\,.\log_2 n] \qquad (5.146\,b)$$

unde prima paranteză pătrată este numărul de operații normale pe un calculator serial*), iar a doua ia în considerare efectul implementării pe un calculator paralel. Observăm că prima paranteză are un minim pentru $l=\log_2(\log_2 n)$, dar a doua crește monoton cu l. Astfel, pentru calculatoare paralele care au $n_{1/2} > 0$ $t_{SERIFACT}$ minim se obține cu l mic, așa cum s-a spus.

Linia de performanță egală între algoritmul cu l nivele de reducere și cel cu $l+1$ se află cu

$$n_{1/2}/n = [(1+5\log_2 n)2^{-(l+1)} - 4]/(4 + 8\times 2^l - 5\times 2^{-(l+1)}) \qquad (5.147)$$

Forma acestei ecuații sugerează că un plan al parametrilor adecvat pentru analiza **SERIFARC** este planul fazelor $(n_{1/2}/n, n)$, lucru arătat în fig. 5.26. Liniile de performanță egală definite de ecuația (5.147) împart planul în regiuni în care $l = 0, 1, 2, 3$ sînt alegeri optime. Liniile de valoare constantă ale lui n_{12} se află la 45° față de axă, iar liniile pentru $n_{12} = 20$, 100, 2048 apar întrerupte în figură. Se consideră că aceste linii sînt tipice pentru calculatoarele **CRAY-1**, **CYBER** 205 și corespund performanței medii a lui ICL DAP. Pentru dimensiuni reale ale caroiajelor (de exemplu, $n < 500$) se indică folosirea lui $l = 1$ sau 2 pe **CRAY-1**, $l = 0$ sau 1 pe

---

* Hockney (1970, 1980) a stabilit valoarea de 4,5 l pentru primul term en din paranteză. Explicația constă în aceea că s-a considerat pentru rezolvarea sistemelor tridiagonale, reducerea ciclică scalară (6 operații pentru un punct), în loc eliminării gausiene considerată aici (5 operativ pentru un punct). Alte considerente au influențe minore asupra acestei ecuații.

CYBER 205 și $l = 0$ pe **ICL DAP**. Valoarea mai mică din cele două ale lui l se aplică problemelor cu $n < 100$. Temperton (1979 b) a determinat timpul de execuție al unui program **SERIFACR**(l) pe **CRAY-1** și a măsurat valoarea optimă $l = 1$ pentru $n = 32$, 64 și 128. Aceste rezultate concordă cu cifrele noastre cu excepția lui $n = 128$, pentru care fig. 5.26 furnizează valoarea optimă $l = 2$. Această discrepanță apare probabil deoarece Temperton folosește forma Buneman a reducerii ciclice (vezi Hockney 1970) care crește costul calculului reducerii ciclice și tinde să deplaseze valoarea optimă a lui l la valori mai mici. Pentru o anumită dimensiune a problemei (valoare a lui n), fig. 5.26 ilustrează mai mult calculatoare seriale ($n_{1/2}$ mic) la stînga și mai mult calculatoare paralele ($n_{1/2}$ mare) la dreapta. Deci, cu cît este mai paralel calculatorul, cu atît este mai mică valoarea optimă a lui l.

În cazul algoritmului **SERIFACR**, vectorii sînt organizați de-a lungul unei direcții a caroiajului și nu depășesc niciodată lungimea n. Acesta este un algoritm adecvat pentru calculatoare care operează bine cu astfel de vectori, acei cu $n_{1/2} < n$ și/sau care au un paralelism natural (sau lungime a vectorului) mai mic sau egal cu n. Ultima afirmație se referă la faptul că unele calculatoare (**CRAY X-MP**, de exemplu) au registre vectoriale capabile de a memora vectori de o anumită lungime (64 la **CRAY X-MP**). De aceea, un algoritm care folosește vectori cu această lungime este avantajos deoarece corespunde hard-ului calculatorului. De exemplu, algoritmul **SERIFACR** va fi adecvat în particular pentru rezolvarea problemei Poisson 64 × 64 pe **CRAY X-MP** cu vectori de lungime maximă 64, deoarece această mașină realizează mai mult de 80% din performanța sa pentru vectori cu această lungime. Pe alte calculatoare, ca CYBER 205, nu există registre vectoriale și $n_{1/2} \approx 100$. Pentru aceste mașini este de dorit ca lungimea vectorului să fie cît mai mare, preferabil de ordinul miilor. Aceasta înseamnă că algoritmul FACR trebuie implementat astfel încît paralelismul să fie proporțional cu n^2 mai mult decît cu n. Cu alte cuvinte,

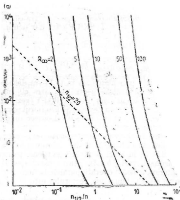


Fig. 5.26 Planul parametrilor ($n_{1/2}/n$, n) pentru algoritmul SERIFACR (l). Liniile pline delimitează regiunile unde valorile lui conduc la timpi de execuție minimi. Liniile întrerupte corespund la $n_{1/2}$ constant pentru CRAY-1 ($n_{1/2} = 20$), CYBER 205 ($n_{1/2} = 100$) și performanța medie a lui ICL DAP ($n_{1/2} = 2048$). (Conform Hockney (1983)).

vectorii corespund structurii bi-dimensionale ca ansamblu, mai degrabă decît după o dimensiune. Algoritmul **PARAFACR**, care va fi descris în continuare, este proiectat tocmai pentru a realiza acest lucru.

(*ii*) *Algoritmul* **PARAFACR**

Fiecare etapă a algoritmului **FACR** poate fi implementată cu vector de lungime proporţională cu n^2.

(a) *Modifică RHS* — la fiecare nivel, r, de reducere ciclică, modificarea părţii din membrul drept poate fi făcută în paralel cu toate cele $n^2 \cdot n^{-r}$ puncte. Deci, formula de evaluare a timpului de execuţie devine

$$t_a = \sum_{r=1}^{l} (n_{1/2} + n^2 \cdot 2^{-r})(3 \times 2^{r-1} + 2) \qquad (5.148\ a)$$

(b) *Analiza Fourier* — cele $n2^{-l}$ transformate de lungime n sînt executate în paralel ca la **SERIFARC**, dar acum folosim un algoritm paralel, **PARAFT**, pentru execuţia FFT cu lungimea vectorului n. Pentru toate liniile lungimea vectorului devine $n^2 . 2^{-l}$ iar ecuaţia timpului devine

$$t_b = (n_{1/2} + n^2 2^{-l})\, 4 \log_2 n \qquad (5.148\ b)$$

Factorul 4 înlocuieşte 2,5 în (5.145 c) deoarece se folosesc operaţii aritmetice suplimentare pentru a păstra cît mai mult lungimea vectorului cît mai mare în algoritmul **PARAFT** (vezi § 5.5.4). Observăm de asemenea că factorul n s-a deplasat în interiorul parantezelor în urma comparaţiei ecuaţiei (5.145 c) cu (5.148 b), deoarece lungimea vectorului a crescut de la $n2^{-l}$ la $n^2 \cdot 2^{-l}$.

(c) *Rezolvarea ecuaţiilor armonice* — ecuaţiile armonice se rezolvă în paralel ca la **SERIFACR** dar folosim o formă paralelă a reducerii ciclice, PARACR, în locul eliminării gausiene, pentru rezolvarea sistemelor tridiagonale (vezi § 5.4.3). Pentru cazul special al coeficienţilor specificaţi anterior, există 3 operaţii paralele la fiecare din cele $\log_2 n$ nivele ale reducerii ciclice. Lungimea vectorului este $n^2\, 2^{-l}$ de unde

$$t_c = (n_{1/2} + n^2 2^{-l})\ \ 3\ \log_2 n \qquad (5.148\ c)$$

(d) *Sinteza Fourier* — ca etapă (b)

$$t_d = (n_{1/2} + n^2 2^{-l})\, 4 \log_2 n \qquad (5.148\ d)$$

(e) *Completarea* —la fiecare nivel r, trebuie rezolvate $n2^{-r}$ sisteme tridiagonale de lungime n. Folosim PARACR, ca la etapa (c), lungimea vectorului este $n^2 \cdot 2^{-r}$. După aceea, pentru fiecare punct se execută 2 operaţii, care pot fi realizate în paralel, obţinîndu-se

$$t_e = \sum_{r=1}^{l} (n_{1/2} + n^2 2^{-r})(3 \times 2^{r-1} \log_2 n\ + 2) \qquad (5.148\ e)$$

Făcînd suma, găsim că pentru fiecare punct timpul de execuție al algoritmului **PARAFACR** este proporțional cu

$$n^{-2}t_{PARAFACR} = s + (n_{1/2}/n^2)q'' \qquad (5.149\ a)$$

unde

$$s = \frac{1}{2}(3\log_2 n + 1)l + 4 + (11\log_2 n - 4)2^{-l}$$

$$q'' = 4l + (3\log_2 n + 1)(2^l - 1) + 11\log_2 n \qquad (5.149\ b)$$

Linia de performanță egală între algoritmul de nivel l și l + 1 este definită de

$$n_{1/2}/n^2 = \left[(11\log_2 n - 4)2^{-(l+1)} - \frac{1}{2}(3\log_2 n + 1)\right]/[4 + (3\log_2 n + 1)2^l]$$

(5.149 c)

Forma ecuației (5.149 c) ne face să alegem pentru reprezentare rezultatele algoritmului **PARAFACR** în planul parametrilor $(n_{1/2}/n^2, n)$, lucru realizat în fig. 5.27. Obținem că liniile de performanță egală sînt aproximativ verticale și concluzionăm că $l = 2$ este optim pentru $n_{1/2} < 0{,}1\, n^2$, $l = 1$ pentru $0{,}1\, n^2 < n_{1/2} < n^2$ și $l = 0$ pentru $n_{1/2} > n^2$. Nu există circumstanțe cînd mai mult de două nivele să fie avantajoase, ceea ce justifică alegerea noastră pentru algoritmul **FACR** nestabilizat. În particular, pentru un masiv de procesoare cu un număr de procesoare mai mare sau egal cu numărul de puncte $(N \geqslant n^2)$, luăm $n_{1/2} = \infty$ și găsim $l = 0$. Acest caz corespunde rezolvării unei probleme 64×64 pe **ICL DAP**, care este un masiv de 64×64 procesoare. Linia întreruptă pentru $n_{1/2} = 100$ este prezentat în fig. 5.27, corespunzător lui **CIBER** 205. Pentru toate caroiajele, mai puțin cele mai mici, (de exemplu, pentru $n \geqslant 30$) găsim $l = 2$ optim). S-a reprezentat și linia pentru $n_{1/2} = 20$, de unde concluzionăm că $l = 2$ este optim în toate circumstanțele, dacă acest algoritm este executat pe CRAY-1.

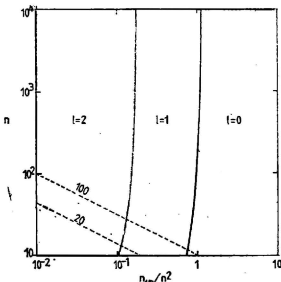


Fig. 5.27 Planul parametrilor $(n_{1/2}/n, n)$ pentru algoritmul PARAFACR (1)). Se folosesc aceleași notații ca în fig. 5.26 (Conform Hockney (1983)).

(iii) *Comparație SERIFACR/PARAFACR*

Pînă acum am considerat alegerea celei mai bune valori a lui l pentru fiecare algoritm. Optimizînd fiecare algoritm, să vedem care este cel mai avantajos de utilizat. Putem afla acest lucru prin trasarea $t_{SERIFACR}$ și $t_{PARAFACR}$ funcție de $(n_{1/2}/n)$, pentru valori diferite ale lui n. Astfel se

determină aproximativ cel mai bun algoritm în regiuni diferite ale planului parametrilor. Se poate calcula apoi linia de performanţă egală între PARAFACR (l) şi SERIFACR (l') din

$$n_{1/2}/n = (a-b)/(c-d) \qquad (5.150\ a)$$

unde

$$a = \frac{1}{2}(3\log_2 n + 1)\,l + 4 + (11\log_2 n - 4)\,2^{-l}$$

$$b = 4l' + 4 + (1 + 5\log_2 n)\,2^{-l'}$$

$$c = 4l' - 8 + 8 \times 2^{l'} + 5 \times 2^{l'} + 5\log_2 n$$

$$d = [4\,l + (3\log_2 n+1)(2^l-1) + 11\log_2 n]/n$$

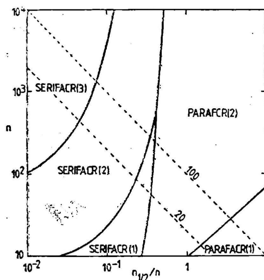


Fig. 5.28 Comparaţie între SERIFACR (1) şi PARAFACR (1), ce prezintă regiunile din plan parametrilor ($n_{1/2}/n^2$, n) unde fiecare se execută într-un timp minim. (După Hockney (1983)

Interacţiunea celor doi algoritmi este prezentată în fig. 5.28, în planul parametrilor ($n_{1/2}/n$, n). Cele doă zone ale planului sînt separate de o linie verticală, unde lSERIFARC este cel mai bun algoritm pentru $n_{1/2}$ mici (sub 0,4 n, calcuatoare mai seriale) şi că **PARAFACR** este cel mai bun pentru $n_{1/2}$ mare (mai mare ca 0,4 n, calculatoare mai paralele). Liniile de $n_{1/2}$ constante corespund calculatoarelor **CRAY**-1 şi CYBER- 205. În concluzie, **SERIFACR** ar trebui folosit pe **CRAY**-1, exceptînd structurile mici cu $n < 64$ cînd **PARAFACR**(2) este probabil să fie cel mai bun. Pe **CYBER** 205 se preferă **PARAFACR**, exceptînd structurile foarte mari cînd **SERIFACR**, exceptînd structurile foarte mari cînd **SERIFACR**(2) ($300 < n < 1500$) sau **SERIFACR**(1) ($n > 1500$) este cel mai bun.

5.6.3 Metode tri-dimensionale

Numărul strategiilor diferite de rezolvare a ecuaţiilor tridimensionale echivalente (5.119 b) este mare. Aici vom indica numai posibilităţile evidente şi vom lăsa utilizatorului estimarea procedurii celei mai bune pentru obiectivul său folosind rezultatele din § 5.6.1. şi § 5.6.2.

Metodele iterative vor trebui considerate numai pentru cazul coeficienţilor generail. Metoda SOR se poate evident generaliza şi la trei dimen-

siuni. Şi metoda SLOR de relaxare a liniei poate fi folosită în trei dimensiuni prin presupunerea că toate liniile învecinate sînt corecte şi prin efectuarea unei corecţii a liniei. Ajustarea unui întreg plan de valori, presupunînd că planele vecine sînt corecte (de exemplu SPOR) poate fi executată dacă există un program pentru rezolvarea planului de valori. Aceasta este, de exemplu, problema bi-dimensională din § 5.6.1 şi § 5.6.2, pentru care trebuie selectată o metodă adecvată. Pentru coeficienţi generali se va alege o metodă iterativă, sau o metodă directă dacă problema bi-dimensională este ecuaţia Poisson bi-dimensională. Evident, alegerea cea mai bună va depinde de coeficienţii ecuaţiei diferenţiale. Există un avantaj în rezolvarea cît mai multor dimensiuni ale problemei prin metode directe. Aceasta va fi posibilă pentru oricare dimensiune care are numai derivate pare calculate pe o structură uniformă.

În discutarea metodelor directe în trei dimensiuni ne limităm la rezolvarea ecuaţiei Poisson discrete [generalizarea ecuaţiei (5.131) la trei dimensiuni]. Următoarea posibilitate se sugerează:

(a) *MFT* — metoda transformatei Fourier multiple definită de (5.133) sau (5.139 b) poate fi aplicată dacă o transformată tridimensională înlocuieşte transformata bi-dimensională discutată anterior. Această metodă este folosită de algoritmul P^3M descris de Hockney şi Eastwood (1988). Metoda nu are un timp de execuţie minim, dar ar putea fi folosită dacă alte circumstanţe cer o cunoaştere a spectrului undelor tridimensionale ale sursei. Aplicaţii posibile sînt reprezentate de fizica plasmei.

(b) *FACR(l)* — această metodă se genereralizează la trei dimensiuni dacă se înlocuieşte transformata Fourier după direcţia x cu o transformată bi-dimensională în planul (x, z). Apoi, se execută reducerea ciclică după direcţia y. Deoarece, în această situaţie, transformata Fourier este de două ori mai costisitoare pentru fiecare punct (datele trebuie procesate după ambele direcţii, x şi z), valoarea optimă a lui l va fi mai mare decît cea din cazul bi-dimensional. Kascic (1983) a adoptat o procedură FACR(0) pentru a rezolva o problemă 64^3 pe **CYBER** 205, deşi el a folosit o formă a decompoziţiei **LU** pentru rezolvarea sistemelor tridiagonale.

(c) *Transformata 1D* — dacă se execută o transformată Fourier unidimensională, să spunem în direcţia z, fiecare armonică respectă o ecuaţie Helmholtz discretă ($\nabla^2\Phi - k\Phi = f$) în celelalte două dimensiuni. Ea poate fi soluţionată de orice rezolvitor Helmholtz. Dacă se întîmplă să fie procedura **FACR** (1) cu transformata Fourier în direcţia x şi reducere ciclică în direcţia y, se reinventează de fapt opţiunea (b) de mai sus.

Cînd se implementează opţiunile anterioare pe anumite calculatoare, trebuie luat în considerare paralelismul n, n^2 sau n^3. În cazul cel mai simplu al MFT, se pot transforma n linii de lungime n în paralel, folosind cel mai bun algoritm serial şi acest proces se va repeta de n ori în cele trei direcţii. Altfel, se pot transforma n^2 linii de lungime n în paralel, folosind cel mai bun algoritm serial, proces repetat după toate direcţiile. În sfîrşit, se pot transforma n^2 linii în paralel, folosind algoritmul paralel PARAFT şi să se obţină un paralelism n^3. Consideraţii similare se pot face referitor la nivelele de paralelism folosite la implementarea metodelor iterative. Evident, nu avem posibilitatea de a prezenta şi compara aici aceste alternative; cu informaţiile prezentate în acest capitol, cititorul interesat poate întreprinde această analiză singur.

6

TEHNOLOGIA ŞI VIITORUL

În decursul ultimilor 5 ani, trecuţi de la publicarea primei ediţii a acestei cărţi, s-au înregistrat progrese importante în utilizarea calculatoarelor paralele. Pînă la acea dată, maşinile vectoriale pipeline precum CRAY-1 au dominat în domeniul arhitecturilor de supercalculatoare. În anii '80, mulţi producători încearcă să satisfacă cererile de putere de calcul prin multiplicarea numărului de procesoare. Toţi producătorii de calculatoare vectoriale de la sfîrşitul anilor '70 s-au orientat spre obţinerea unor performanţe mai mari prin includerea mai multor procesoare interconectate pirntr-o memorie comună. Un alt progres major este reprezentat de apariţia transputerului INMOS, ca şi a altor soluţii similare implementate pe un cip, care pot fi procesoarele multiple ale unui sistem. De exemplu, transputerul T800 poate lucra continuu cu o viteză de 1Mflop/s şi poate fi conectat cu alte transputere într-o reţea după 4 direcţii. Procesorul N-dimensional (Emmen 1986) poate executa 0,5 Mflop/s şi poate fi conectat într-o reţea după 10 direcţii (hipercuburi cu pînă la 1024 procesoare). Un număr de 1000 de circuite nu reprezintă, în ambele situaţii, un număr excesiv de mare pentru a fi inclus într-un singur sistem şi, de aceea, aceste circuite VLSI CMOS vor concura ca performanţe supercalculatoarele ECL mai costisitoare.

Într-adevăr, este folositor să analizăm aspectele tehnologice ale îmbunătăţirii performanţei supercalculatoarelor. În gama calculatorului CRAY-1, de exemplu, deşi performanţa a crescut în această perioadă de 6 ori, numai o creştere de 1,5 este rezultatul creşterii frecvenţei ceasului; celălalt factor de 4 provine de la utilizarea mai multor procesoare. Este destul de probabil că timpii de propagare pe o poartă s-au micşorat de mai mult de 1,5 ori, decît arată creşterea frecvenţei ceasului, totuşi, lungimea conexiunilor electrice nu a fost redusă semnificativ la aceste maşini. Din acest motiv creşterea frecvenţei ceasului este limitată. Prin folosirea unui sistem de răcire ingenios, CRAY-2 poate lucra cu o frecvenţă mai mare, tocmai datorită compactizării sale. Oricum, tendinţa nu este spre producerea unor maşini ieftine. De aceea, se poate spune că multiplicarea este o soluţie avantajoasă (dorită sau nu) pentru obţinerea unei viteze de calcul mai mare. Unde ne vom îndrepta acum, după ce această barieră a fost depăşită, este subiectul acestui capitol. Ne vom limita la consideraţii

privind arhitectura, deşi nu vom ignora impactul supra algoritmilor şi aplicaţiilor, mai ales că acestea din urmă reprezintă forţacare propulsează cererea continuă pentru o putere de calcul mai mare.

În introducerea acestui volum am văzut că viteza de calcul creşte în medie de 10 ori la fiecare 5 ani (vezi fig. 1.1). Această evoluţie se datorează solicitărilor formulate de utilizatori şi nu există nici un motiv care să o modifice. De fapt, cererea pentru o viteză de calcul mai mare este în creştere. Departamentul apărării al S.U.A. a lansat un program pentru dezvoltarea unor procesoare VLSI care să execute 3×10^9 op./s (Sumney 1980). Alte aplicaţii importante solicită la rîndul lor o viteză de calcul mai mare (Sugarman 1980) multe din ele fiind importante pentru modul nostru de viaţă (de exemplu, modelarea resurselor de energie, vreme şi climă, sau tomografie).

Ca toate realizările noi, utilizarea paralelismului în sistemele de calcul s-a înregistrat la cele mai avansate produse oferite pe piaţă. Calculatoarele ştiinţifice mari sau supercalculatoare, cum sînt acum denumite, sînt costisitoare, dar au performanţele actuale cele mai mari (în mod curent 500 Mflop/s). Oricum, aşa cum s-a întîmplat cu toate produsele rentabile, ele şi-au găsit nişe într-o piaţă mai largă şi au început să fie folosite într-o gamă mult mai variată de aplicaţii.

Aplicaţiile care vor asigura în viitor pieţe mai mari de desfacere s-au îndepărtat de domeniul ştiinţific şi al simulării şi vor corespunde unor aspecte mult mai apropiate de viaţa oamenilor. De exemplu, sistemele expert şi aplicaţiile de baze de date vor fi utilizate mult mai frecvent. Proiectul japonez al generaţiei a cincea, proiectul britanic Alvey şi ESPRIT al Pieţei comune aparţin acestui domeniu de interes. Alte exemple sînt: interfeţele „prietenoase", care realizează înţelegerea limbajului natural şi comunicaţia prin voce; aplicaţiile de prelucrare a imaginilor, pentru birouri sau sisteme de automatizări; aplicaţiile de prelucrare a semnalelor pentru sisteme ca cele de ghidare radio sau radar.

Nu este nici o îndoială că necesarul de viteză de calcul pentru aceste aplicaţii va fi asigurat de calculatoarele paralele. De asemenea, este evident că aceste solicitări sînt încurajate de puterea de calcul ieftină a circuitelor VLSI. De aceea, orientăm acest capitol spre noile rezultate tehnologice, eficienţa lor, ca şi limitelor caracteristice sistemelor paralele şi distribuite descrise în cap. 2 şi 3.

6.1 Caracterizare

Progresele recente înregistrate de proiectarea circuitelor integrate în particular noile nivele de integrare posibile, vor permite implementarea multor idei noi privind aspectele arhitecturale. De asemenea, tehnologia este o componentă cheie a unora din parametrii definiţi. În particular, performanţa specifică a unui calculator paralel, definită în § 1.3.4 ca performanţa pe unitatea de paralelism, depinde foarte mult de tehnologie. În acest caz, influenţa cea mai mare o are viteza circuitelor. Aceasta poate fi măsurată în sistemele bine proiectate prin timpul de propagare pentru

o singură poartă, τ_d. Influențe indirecte intervin în termeni de putere disipată, deoarece aceasta este un factor major ce determină densitatea și nivelele de integrare.

Puterea consumată de o poartă, P_D, este formată dintr-o componentă statică și una tranzitorie. În cazul unei porți active numai „la cerere", componenta statică este foarte mică și poarta consumă energie numai cînd comută. De aceea, necesarul de energie va fi o funcție de frecvența ceasului. Alte circuite sînt „cu pierdere de sarcină" și în acest caz, componenta tranzitorie este mică în comparație cu cea statică. Deoarece consumul de energie variază cu numărul de intrări comandate, de obicei se prezintă un consum de energie mediu.

Pentru orice tehnologie va fi întotdeauna un compromis între puterea consumată și timpul de propagare, deoarece circuitele vor comuta mai rapid dacă sînt comandate de curenți mai mari. Punctul de lucru de pe curba energie consumată-timp de propagare este uneori fixat în cursul procesului de producție, sau poate fi modificat cu rezistențe exterioare circuitului. Produsul $\tau_d . P_D$ este un alt parametru asociat cu o anumită tehnologie și reprezintă o măsură a energiei de comutație pentru o poartă, de exemplu

$$E_{sw} = \tau_d \ . \ P_D \tag{6.1}$$

Pentru a ilustra importanța parametrilor energie de comutație sau consum de energie, vom considera consumul de energie al unui singur circuit. În general, limita de putere consumată pentru un singur circuit integrat este de aproximativ 2W, deși aceasta poate fi cel puțin dublată în cazul unor circuite speciale și posibil răcite cu un agent fluid. Dacă considerăm un consum de 2W pe circuit, cîte porți pot fi integrate? În cazul porților cu pierdere de sarcină aceasta este dată de ecuația (6.2), unde $\overline{P}_D$ este consumul de putere mediu pentru o poartă :

$$N_g \leqslant 2/\overline{P}_D \tag{6.2}$$

În celălalt caz, consumul de putere va depinde de frecvența de comutare medie pe poartă $\overline{f}_c$. Astfel, dacă considerăm două tranziții logice pe ciclu, valoarea limită a lui N_g este furnizată de ecuația (6.3)

$$N_g \leqslant (\overline{f}_c E_{sw})^{-1} \tag{6.3}$$

Frecvența medie de comutare va fi în general mai mică decît cea a sistemului, deoarece nu toate porțile vor efectua tranziții la fiecare tact. Aceasta este important în special pentru tehnologia memoriilor, deoarece în acest caz numai cîteva celule sînt accesate în fiecare ciclu de memorie. Datorită dimensiunilor circuitului, celulele de memorie folosesc de obicei circuite cu pierdere de sarcină. În tab. 6.1 și 6.2 se evidențiază aceste relații.

Mai sînt și alte considerații care limitează numărul de porți pe un singur circuit, cea mai evidentă fiind suprafața porții. Suprafața porții este un factor decisiv pentru costul circuitului, măsurînd necesarul de

siliciu folosit în mod real ('siicon real estate'). Deoarece dimensiunile minime ale dispozitivelor tind spre limita de 1 μm, utilizarea suprafeței este dominată mai mult de conexiuni, așa cum vom vedea ulterior.

Avem în acest moment trei parametri care caracterizează o anumită tehnologie : timpul de propagare pe poartă τ_d, puterea disipată pe poartă. P_D (sau energia de comutare $E_{sw} = \tau_d P_D$) și, în sfîrșit, suprafața porții. În cele ce urmează vom folosi acești parametri pentru aprecierea comparativă a tehnologiilor. De asemenea, vom arăta cum progresele înregistrate de tehnicile de procesare vor influența acești parametri pentru o anumită tehnologie.

Tehnologiile pe care le analizăm aparțin la două categorii : circuite realizate cu tranzistori bipolari și circuite realizate cu tranzistoare cu efect de cîmp, în particular MOSFET. În primul caz, circulația curentului se realizează în plan, în timp ce în cazul tehnologiei bipolare curenții circulă și pe direcție verticală între regiunile dopate cu diverse impurități, așa cum se arată în fig. 6.1. „Electronii liberi" sînt reprezentați de impurități de tip n, în timp ce „golurile libere" sînt create de impurități p. Golurile reprezintă absența electronilor, care se deplasează și creează curenți în același mod cu electronii liberi dar, desigur, în direcție opusă. Ele sînt, oricum, mai puțin mobile. Deoarece granițele între aceste regiuni pot fi controlate cu precizie mare ($\sim$ 0;1 μm) în comparație cu dimensiunile planare (0,1 m), tehnologiile bipolare tind să fie mai rapide. Circuitele realizate sînt însă mai complexe.

Între aceste două categorii se află multe tehnologii diferite, funcție de circuite, materiale și caracteristicile de procesare. De exemplu, multe din tehnologiile experimentale se bazează pe noi modalități de prelucrare. Acestea reduc dimensiunea dispozitivului și, prin urmare, reduc putere consumată și cresc viteza.

Tabelul 6,1 **Nivele posibile de integrare pentru energii diferite de comutare și frecvențe de ceas pentru tehnologia porții „la cerere".**

f_c (MHz)	E_{sw} (pJ)					
	10^{-1}	1	10	10^2	10^8	10^2
10^{-1}	10^8	10^7	10^6	10^5	10^4	10^3
1	10^7	10^6	10^5	10^4	10^3	10^2
10	10^6	10^5	10^4	10^8	10^2	10
10^2	10^5	10^4	10^3	10^8	10	1
10^8	10^4	10^3	10^2	10	1	—

VLSI LSI MISI SSI

Tabelul 6.2 **Nivele posibile de integrare pentru tehnologia porții „cu pierdere de sarcină" ca o funcție de putere medie disipată pe poartă.**

P(mW)	200	20	2	0,2	0,02
N_g	10	10^2	10^3	10^5	10^4

SSI MSI LSI VLSI

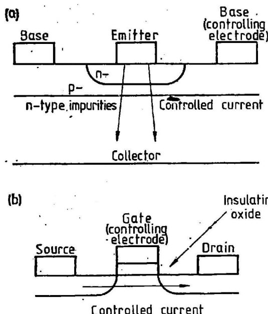


Fig. 6.1 Secțiune transversală prin doi tranzistori de comutare elementari: (a) tranzistorul bipolar; (b) MOSFET

6.2 Tehnologii bipolare (TTL, ECL, I²L)

Cele trei tehnologii bipolare majore sînt transistor-transistor-logic (TTL), emitter coupled logic (ECL) și integrated injection logic (I²L). Uneori ECL mai este denumită și current mode logic (CML), iar I²L merged transistor logic (MTL).

Primul circuit integrat realizat a fost TTL. În anul 1964, Texas Instruments a anunțat prima familie de circuite TTL. NAND este poarta elementară, care lucrează cu nivele de tensiune de 0 și 2 V, pentru valorile logice 0, respectiv 1. TTL este lentă și consumă multă energie; de aceea, acum este depășită. Totuși, se mai folosesc seriile TTL, denumite Schottky TTL, low-power Schottky TTL și FAST, în aplicații pe scară mică/medie.

Piața circuitelor TTL este în declin; tehnologiile care se impun oferă circuite proiectate conform prescripțiilor cumpărătorului (customisable chips), care au o densitate mare de integrare. O soluție ieftină o reprezintă PAL, masiv logic programabil (field programmable logic array), înscris într-o manieră similară circuitelor EPROM. Dacă sînt necesare circuite mai complexe, piața circuitelor de tip masiv poate oferi soluția. Acestea sînt circuite produse special pentru anumiți cumpărători (user-customisable), de obicei în tehnologie CMOS, dar și bipolară, care permite ca ultima etapă de fabricare să se realizeze conform specificațiilor utilizatorului. Aceste circuite conțin 500—20 000 porți logice, aranjate într-o structură regulată de masiv, iar funcțiile cerute de utilizator sînt create prin conexiuni metalice între porțile corespunzătoare. De obicei, acest proces se realizează automat și se poate executa după o schemă elaborată de calculator. Șabloamele de conectare pot fi implementate foarte rapid folosind fluxuri de electroni. Ansamblul acestor tehnici este ieftin și cel puțin o companie își oferă serviciile în această direcție la prețul de cîteva sute de lire sterline.

O altă familie de circuite logice bipolare este ECL, cea la care tranzistorii nu intră în regim de saturație. Circuitul de bază ECL este inversorul. Acesta poate fi folosit pentru crearea porții duale OR/NOR. Valorile logice 0 și 1 sînt reprezentate de —1,7 V și —0,8 V. Adesea circuitele ECL sînt produse pentru a respecta anumite condiții. Aceasta nu înseamnă

că circuitele vor fi realizate în întregime conform specificațiilor beneficiarului, deoarece tehnologia masivelor de porți OR/NOR este acum standard.

Principalul avantaj al tehnologiei ECL este timpul de propagare pe poartă foarte mic, care poate fi de 100 ps. Alt avantaj este fan-out-ul mare, pentru care totuși se plătește o penalitate prin scăderea vitezei. Dezavantajele sînt consumul mare de energie, în jur de 1mW pe poartă pentr timpi de propagare sub ns și suprafața relativ mare a porții ($> 300\ \mu m^2$). De aceea, în mod obișnuit ECL nu este un candidat bun pentru circuitele LSI. Capsulele obșnuite conțin în jur de 1000 porți.

6.3 Tehnologii MOS (NMOS și CMOS)

Tranzistorul cu efect de cîmp (FET) este un dispozitiv relativ simplu (fig. 6.1 (b),, în care un cîmp generat de un electrod al porții controlează circulația curentului într-un canal între sursă și drenă. Cel mai obișnuit FET este denumit MOSFET, după electrodul porții care are o structură sandwich metal-oxid-semiconductor. De curînd, metalul a fost înlocuit de polysilicon ceea ce face să fie posibil un al doilea nivel de interconectare. Canalul este format fie din sarcini n, fie p, de unde tehnologia rezultată NMOS, sau PMOS. Deși NMOS se mai utilizează încă, a treia tehnologie MOS importantă folosește atît tranzistoare NMOS cît și PMOS. Această tehnologie numită CMOS este convenabilă pentru implementarea VLSI și are proprietăți de disipare a energiei foarte favorabile.

Avantajele principale ale tuturor tehnologiilor MOS sînt modalitățile de producere relativ simple și densitățile mari de integrare. Astfel, MOS a fost folosit aproape exclusiv pentru aplicațiile LSI, furnizînd produse foarte ieftine (de exemplu, microprocesoare și circuite de memorie de mare capacitate). Dezavantajul circuitelor MOS este viteza de lucru scăzută , deși, în acest domeniu se lucrează intens. Circuitele CMOS obișnuite vor lucra la aproximati 20—40 MHz.

Poarta elementară NMOS este inversorul, care folosește numai doi tranzistori, din care unul este întotdeauna deschis. Astfel, acest circuit lucrează ca o rezistență cu valoare fixă mare, în timp ce al doilea lucrează ca un comutator (fig. 6.2). Modul de lucru al porții este simplu : cele două tranzistoare reprezintă o rețea de rezistențe, ce divizează tensiunea între sursă și masă. Cînd T_1 este deschis ($V_{in} \approx 5V$), V_{out} are aproximativ valoarea nulă ($V_{out} \approx 0V$) și reciproc. Cu ajutorul unor perechi de tranzistori de comutație (T_{1A} și T_{1B}) conectați fie în serie, fie în paralel la masă (fig. 6.3) se pot construi porți NAND și NOR.

Datorită valorilor mari ale rezistențelor, tehnologiile MOS consumă foarte puțină energie. De exemplu, cînd T_1 este deschis ($V_{in} \approx 5V$), rezistența totală dintre sursă și masă este în mod obișnuit de $100\ k\Omega$. Astfel, pentru o sursă de 5V

$$P_D = 0{,}25\ \ mW$$

iar cînd T_1 este închis

$$P_D = 2{,}5\ nW$$

Se observă, în acest mod, că porţile NMOS consumă o cantitate semnificativă de energie numai cînd ieşirea lor este „low"; nici vîrfurile de putere tranzitorii nu sînt semnificative.

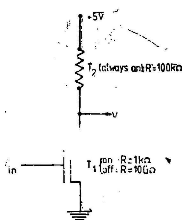


Fig. 6.2 Circuitul unui inversor MOS

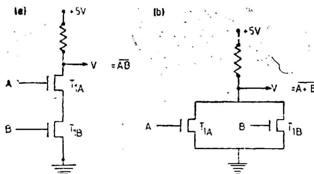


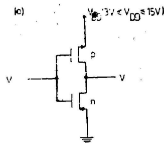
Fig. 6.3 Combinarea tranzistorilor MOS pentru realizarea funcţiilor logice duale (a) poarta NAND; (b) poarta NOR.

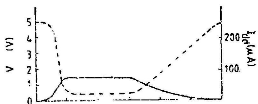
Densitatea circuitului NMOS cu porţi inversoare ce respectă regula 2 μm este de aproximativ 25 000 porţi pe mm^2 funcţie de geometria inversorului, dar în practică această valoare este rar întîlnită.

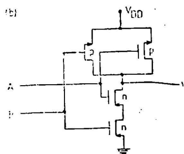
La circuitele NMOS există o asimetrie între timpii de ridicare şi cădere, datorită tranzistorului T_2 care lucrează ca o rezistenţă pasivă. Căderea semnalului este activă şi necesită în mod obişnuit 1—5 ns, în timp ce creşterea semnalului este pasivă şi durează 4—20 ns. Fenomenul este prezentat în fig. 6.5 (a), unde se arată caracteristicile tensiunii tranzitorii şi a curentului. (Observaţie: timpul de propagare este foarte sensibil la încărcarea porţii, la tehnologiile MOS: valorile prezentate se referă la porţi încărcate corect).

Tehnologia CMOS evită timpii lungi de stabilire a semnalelor. Deoarece PMOS are polaritatea opusă lui NMOS, dispozitivele respective pot fi folosite în perechi complementare pentru a forma porţile inversoare ele-

mentare (vezi fig. 6.4 (a)). Se formează un inversor aproape simetric (există diferențe în mobilitatea purtătorilor) în care numai o poartă este activă la un moment dat. Porțile NAND și NOR sînt construite folosind două perechi complementare de tranzistoare. Poarta NAND, arătată în fig. 6.4(b)







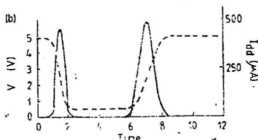


Fig. 6.4 Circuite logice complementare MOS (CMOS): (a) inversorul; (b) poarta NAND

Fig. 6.5 Caracteristicile dinamice ale porților de comutare MOSFET: (a) NMOS; (b) CMOS. Curba întreruptă V_{out}, curbele pline I_{DD}

folosește tranzistori de tip p în paralel și de tip n în serie. Poarta NOR este construită cu tranzistoare p în serie și de tip n în paralel.

O altă caracteristică atractivă a tehnologiei CMOS este că se consumă curent numai la comutare (fig. 6.5) (b). Aceasta se verifică ușor deoarece la orice pereche complementară un dispozitiv este întotdeauna inactiv. Astfel, consumul de energie va fi o funcție de frecvență medie a ceasului. Acesta este un mare avantaj pentru tehnologia memoriilor, unde circuitele NMOS obișnuite ating bariera termică de aproximativ 1—2W (Wollesen 1980). CMOS este caracterizată de densități de integrare mai mici decît NMOS și este un proces mai costisitor. Totuși, CMOS este acum mai competitivă și a devenit tehnologia preferată a anilor 1980.

Un dezavantaj al tuturor proceselor MOS, care nu este specific dispozitivelor bipolare, este că nu pot funcționa la cald. La o creștere peste 100°C, dispozitivele MOS își scad viteza cu un factor de doi. Deci, tehnologiile MOS sînt mai mult limitate de puterea disipată de o capsulă de 1—2W, decît tehnologiile bipolare.

6.4 Tehnologii de scalare

Calculatoarele paralele, în special cele cu replicarea resurselor, corespund foarte bine revoluţiei continue care are loc în industria microelectronică. Integrarea pe scară foarte largă (VLSI), cu 10^5 sau chiar 10^6 porţi pe un singur circuit, este acum ceva obişnuit, iar tehnologia marchează noi progrese în domeniul facilităţilor de procesare, în special în privinţa imprimării sau „scrierii'' circuitelor pe feliile de siliciu.

Efectele micşorării dimensiunilor dispozitivelor au fost un subiect de mare interes şi, pentru tranzistoarele MOS, se cunosc reguli care menţin proprietăţile dispozitivelor (Dennard et al 1974, Hoeneisen şi Mead 1972). Pentru un factor de scalare de k (> 1), dacă toate distanţele orizontale şi verticale sînt micşorate de 1/k ori, la fel nivelele de dopare, atunci utilizînd nivele de tensiune micşorate de 1/k ori, şi caracteristicile dispozitivelor se vor micşora conform tabelei 6.3 (Hayes 1980).

Aşa cum am văzut, este de dorit să se micşoreze timpii de propagare, dar cu efectul creşterii puterii disipate. Aplicînd regulile de scalare, timpul de propagare se micşorează de 1/k ori, iar puterea consumată de $1/k^2$ ori, ambele efecte fiind foarte favorabile. Observăm că densitatea de integrare creşte de $1/k^2$ ori, iar densitatea puterii rămîne aceaşi. Astfel este posibil să se crească densitatea de integrare prin scalarea dispozitivelor, fără a se atinge bariera energiei termice disipate.

Scalarea pune şi probleme, deoarece densitatea curentului creşte printr-un factor de k, ceea ce poate influenţa fiabilitatea capsulei. Dacă densitatea curentului devine prea mare, şi conexiunile metalice vor trebui să aibă o suprafaţă mai mare. Alte probleme intervin datorită micşorării tensiunilor, şi în mod corespunzător a diferenţei între nivelele logice. Nivelele de zgomot rămîn constante datorită energiei termice a particulelor discrete, ceea ce poate constitui o problemă critică. Micşorarea dispozitivelor, fără scăderea corespunzătoare a tensiunilor conduce la o creştere de k^3 a densităţii puterii.

Tabelul 6.3 **Factori de scalare caracteristici pentru o tehnologie MOS, căreia i se aplică factorul de scalare K^{-1} după dimensiunile orizontală şi verticală, ca şi tensiunilor, şi cu densitatea impurităţilor micşorată de K ori.**

Caracteristica	Factorul de scalare
Curentul I	K^{-1}
Timp de propagare τ_d	K^{-1}
Puterea disipată P_D	K^{-2}
Produsul $\tau_d P_D$	K^{-3}
Rezistenţa R	K
Densitatea curentului J	K
Densitatea de încapsulare	K^{-2}
Densitatea de putere	1
Factorul IR	1
Constanta RC	1

O ultimă problemă privind micșorarea dimensiunilor dispozitivelor intervine prin creșterea dimensiunii relative și a întîrzierii de propagare relativă a dispozitivelor care comunică cu mediul extern. De aceea, întregul beneficiu al scalării poate fi regăsit în interiorul capsulei, iar atunci cînd se transmit semnale în exterior el se micșorează.

Deși scalarea este foarte atractivă, și pare a fi nelimitată, există anumite limitări fundamentale datorită naturii fizice a dispozitivelor. Mead și Conway (1980) tratează în detaliu aceste aspecte. În practică nu se tinde spre atingerea lor, atît timp cît dimensiunile nu scad sub 0.5 μm.

Prin contrast, scalarea tranzistorilor bipolari nu este nici atît de riguroasă, nici atît de bine definită. Tranzistorii bipolari pot fi micșorați fără a modifica a treia dimensiune; într-adevăr, cu tehnologiile existente este foarte dificil de micșorat și această dimensiune. Deoarece timpii tranzistorii depind la tranzistorii bipolari de dimensiunea verticală, timpul de propagare ECL nu se va micșora liniar odată cu dimensiunile planare. O reducere se va înregistra datorită efectelor capacitive, iar micșorarea puterii va fi similară cu cea de la dispozitivele MOS.

O tratare a simulării scalarii atît a dispozitivelor bipolare, cît și MOS, a realizat Hart et al (1979). Rezultatele obținute de ei sînt prezentate în fig. 6.6 și 6.7. Dispozitive submicronice experimentale confirmă rezultatele simularilor (de exemplu, Fang și Rupprecht 1975, Sakai et al 1979). Cît de curînd se vor produce aceste dispozitive, este mai dificil de răspuns. Regulile de scalare aparent simple presupun îmbunătățiri ale tehnologiilor de prelucrare și există o mare diferență între obținerea experimentală și în producție a unor astfel de dispozitive VLSI.

6.5 Problemele puse de scalare

Am văzut în capitolele 2, 3 și 5 că atît la proiectarea calculatoarelor, cît și a algoritmilor o problemă majoră este cea a comunicării. Și în secțiunea anterioară am făcut unele aluzii la problemele de comunicație ce apar la nivelul capsulei. Cu alte cuvinte, problema comunicației nu este ceva care să dispară automat prin folosirea baghetei magice reprezentată de tehnologia VLSI; accentul ei este mai degrabă deplasat de la nivelul sistemului la cel microelectronic. În acest mod, comunicația rămîne cheia unui proiect de calculator reușit.

Comunicația este, într-un calculator electronic digital, propagarea unor semnale electrice de frecvență mare, de-a lungul unor fire sau circuite imprimate, sau prin structurile de siliciu dopat cu impurități. Faptul că aceste semnale se propagă într-un interval de timp finit între diversele componente ale unui calculator, și că acest interval poate fi semnificativ, a fost descoperit încă de la primele proiecte MU. Universitatea Manchester (MU) a întîlnit probleme serioase la construirea primului calculator ATLAS. De atunci structura fizică a calculatorului a devenit din ce în ce mai importantă și este acum un aspect important al proiectării.

CRAY-1 rezuma această limită de proiectare deoarece toate aspectele mașinii sînt proiectate în sensul minimizării și egalizării timpului de propagare. Caracteristica cea mai evidentă este forma izbitor de neobișnuită

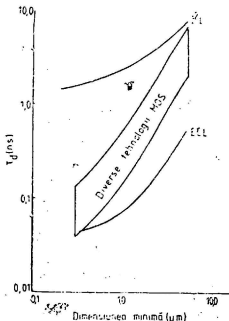


Fig. 6.6 Simularea scalării timpului de propagare pe o poartă pentru ECL, I^2L şi diverse tehnologii MOS (date luate din lucrarea lui Hart et al 1979).

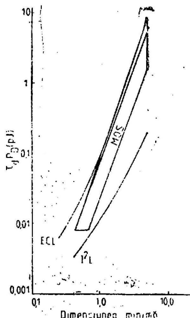


Fig. 6.7 Simularea scalării produsului putere timp de propagare pe poartă pentru ECL, I^2L şi diverse tehnologi i MOS (după Hart et al 1979).

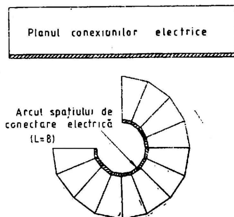


Fig. 6.8 Reducerea lungimii conexiunilor, folosind o organizare de suprafaţă constantă

a carcasei (vezi fig. 2.1). Aceasta poate fi considerată o transformare topologică a unei carcase paralelipidedice pentru a reduce lungimea firelor de legătură. Fig. 6.8 prezintă o transformare de arie egală a suprafeţei carcasei calculatorului CRAY. Se poate observa că o dimensiune a planului de conexiuni s-a redus la jumătate. Forma ideală este cea sferică, cu simetrie completă care minimizează lungimea conexiunilor electrice. Probabil CRAY-2 va avea această formă!

Alte caracteristici ale calculatorului CRAY-1 care minimizează timpul de propagare sînt liniile de transmisie terminate cu rezistenţe şi densităţile de integrare pe capsulă extrem de ridicate. Datorită lor se pun probleme deosebite de răcire, mai mult de 100 kW de putere disipată termic trebuie eliminată dintr-un volum de aproximativ 34 m^2. Pentru a înţelege amploarea problemei, să ne închipuim un element electric de 1 kW

aflat într-un recipient de cositor care trebuie menţinut la temperatura camerei. Chiar şi în aceste condiţii asigurate de sistemul de răcire, 50% din perioada ceasului de 12,5 ns este urmare a timpilor de propagare.

Pentru a înţelege problemele de scalare, să considerăm că: într-o linie de transmisie perfectă semnalele se propagă cu viteza luminii, 952×10^6 pic/s, ceva sub 1 pic/ns În practică, liniile de transmisie au o anumită capacitate şi rezistenţa, de aceea viteza de propagare a semnalelor va fi mai mică. Pentru o reţea RC, propagarea semnalelor este descrisă de ecuaţia de difuzie:

$$RC \frac{dV}{dt} = \frac{d^2v}{dx^2} \qquad (6.4)$$

unde R şi C sînt rezistenţa şi capacitatea pe unitate de lungime. Timpul de propagare variază pătratic cu lungimea pentru R şi C constante, situaţie caracteristică propagării semnalelor în interiorul capsulei.

Revenind la tabelul 6.3, observăm că, deşi întîrzierea RC rămîne constantă în cursul scalării, rezultatul se bazează pe micşorarea lungimii conexiunilor electrice. Este raţional să presupunem păstrarea dimensiunilor capsulei, pentru a culege roadele densităţii de integrare sporite. În această situaţie distanţele vor fi relativ mai mari, cu rezultatul că traseele electrice din capsulă vor fi mai lente decît factorul de k^2, folosind aceleaşi caracteristici de curent. Într-adevăr, situaţia este chiar mai rea, deoarece acest timp trebuie comparat acum cu timpul de propagare pentru poartă mai mic, scalat de k^{-1} ori, rezultînd o degradare totală de k^3 ori a timpului pentru întreaga conexiune electrică. Deşi efectele nefavorabile ale micşorării dimensiunii abia se pot întrezări, este probabil că acest factor va avea un efect important asupra proiectelor de sisteme ce vor folosi noile tehnologii. Ele vor fi extrem de rapide la nivel local, dar tot mai lente la nivel global sau în exteriorul capsulelor. Să discutăm acum implicaţiile lor.

Cu îmbunătăţirea tehnicilor de prelucrare, tot mai multe funcţii sînt integrate pe un singur cip, iar factorii de scalare descrişi mai înainte au un efect tot mai important asupra arhitecturilor implementate şi metodologiilor de proiectare. Problemele care trebuie depăşite includ:

a) creşterea efortului de proiectare;
b) timpi de propagare pe conexiuni mai mari;
c) densitate mai mare de conexiuni;
d) discrepanţă între performanţele din interiorul şi exteriorul capsulelor.

Ultima situaţie intervine datorită modului de comunicare cu mediul extern prin suduri (pads), pini şi trasee electrice. Apare o discrepanţă între performanţa din interiorul capsulei şi cea din exteriorul ei. Circuitele din interiorul capsulei pot lucra cu o frecvenţă de 25—50 MHz, dar este dificil să se păstreze această frecvenţă la conexiunile externe, fără a creşte curenţii folosiţi.

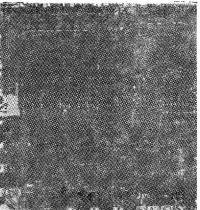
Densitatea conexiunilor poate fi ilustrată prin considerarea efectelor scalării asupra interconectării unor module abstracte din interiorul capsulei. Dacă presupunem utilizarea aceleiaşi scalări ca şi în cazul parametrilor

electrici, se pot obţine k^2 mai multe module pentru o complexitate dată a capsulei. Dacă presupunem că fiecare modul este conectat la toate celelalte, atunci pentru n module sînt necesare m^2 conexiuni. Circuitul micşorat necesită $(nk^2)^2$ conexiuni rezultînd o creştere de k^4 a densităţii conexiunilor pentru o creştere de k^2 a densităţii de module. Aceasta este, evident, o situaţie mai dificilă, deoarece, în general, modulele nu vor fi complet interconectate. Cea mai bună situaţie, cînd se păstrează un echilibru între conexiuni şi densitatea porţilor, va fi aceea cînd este necesar să se interconecteze fiecare modul la un singur alt modul. Aşa cum s-a mai menţionat, problema comunicaţiilor trebuie distribuită între nivelul sistemului şi nivelele de integrare; reţeaua liniară care se va obţine va fi adecvată numai pentru cîteva aplicaţii (vezi §3.3). Această problemă majoră influenţează stilurile diferite de proiectare şi arhitecturile în sensul rezolvării sau ameliorării ei. Soluţiile, ca în cele mai multe probleme inginereşti, constau în abordarea problemei pe toate fronturile, inclusiv (de exemplu) prin introducerea unor noi modalităţi de interconectare prin fibre optice (Goodman et al 1984).

Din punctul de vedere al proiectării se pot găsi modalităţi care să fie folosite atît la nivelul sistemului, cît şi al capsulelor. La nivelul capsulelor, se pot folosi structuri regulate mari de porţi, care elimină problemele ridicate de densitatea conexiunilor. Dimensiunea modulului creşte, iar densitatea se ameliorează prin folosirea unei reţele regulate de conexiuni locale. Iată exemple: RAM; ROM; PLA; masive de comutatoare; matrici de porţi.

Fig. 6.9 care prezintă două microprocesoare, unul din primele şi unul modern, ilustrează cele discutate. La primul, modulul este reprezentat de poartă, iar funcţiile sînt implementate aleatoriu, în modalitatea de lucru cu circuitele TTL. La ultimul, structurarea este mult mai evidentă, incluzînd blocuri RAM, ROM şi căi de date regulate în interiorul ALU.

Deşi RAM şi ROM sînt structuri ideale pentru VLSI, datorită formei lor regulate, numai cu RAM se pot executa puţine operaţii: arhitecturile de procesoare care exploatează de asemenea această structură regulată şi fină sînt candidaţi buni pentru circuitele VLSI. Corespunzător, putem folosi la nivelul sistemului aceste tehnici care creează structuri regulate, de preferinţă locale. Modalităţile de calcul pipeline şi prin multiplicare au fost analizate pe larg, pentru creşterea performanţei maşinii fără a folosi circuite cu viteze mai mari. De aceea este avantajos să le considerăm acum prin prisma unor structuri regulate cu comunicaţii locale. Fig. 6.10 prezintă o cale de date pipeline, unde se pot observa că registrele introduse pentru memorarea rezultatelor parţiale creează regiuni temporale locale, în locul uneia generale. Semnalele de ceas necesare pentru sincronizarea circulaţiei datelor prin pipeline sînt în continuare semnale generale; oricum întîrzierea cu care se propagă nu prejudiciază funcţionarea sistemului, avînd în vedere că timpii de propagare la module diferite pot fi egalizaţi. Dacă este necesar, sistemul poate lucra asincron, sau cu sincronizare locală, prin stabilirea unei modalităţi de preluare a datelor între etaje. Între modulele adiacente se va implementa un protocol de handshake. În § 4.5.2 se descrie tehnica de programare cu pipeline asincron-



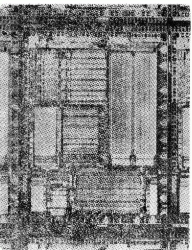


Fig. 6.9 Comparaţie între diferite generaţii de microprocesoare. (a) Intel 8080, microprocesor pe 8 biţi. (b) Un transputer T800, microprocesor pe 32/64 biţi.

Fig. 6.11 prezintă structura la nivelul cipului a unui masiv de procesoare tipic, care folosește pentru creșterea performanțelor principiul multiplicării. Din nou se poate observa că s-au ales între modulele multiple conexiuni care să reflecte natura planară a mediului și deci locală. Aceleași

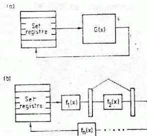


Fig. 6.10 O diagramă ce compară un ALU convențional (a) cu un ALU pipeline (b). Se observă că ultima are comunicații distribuite local.

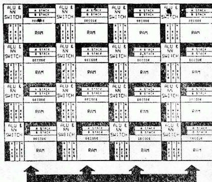


Fig. 6.11 Diagrama ce ilustrează structura și comunicațiile regulate locale ale unui element procesor RPA.

argumente privind controlul se aplică și structurilor de masive de procesoare. Se poate folosi un sistem sincron, în care caz impulsurile și chiar cuvîntul de control vor fi furnizate de o sursă unică. Altă modalitate este ca sistemul să fie cu control local, în care caz fiecare procesor va avea pro-

priul ceas şi mediu de memorare local. Prima soluţie este tipică pentru calculatoarele SIMD, iar ultima ar corespunde mai multor transputere realizate pe o capsulă. INMOS a abordat astfel de soluţii, care ar putea fi generate automat de către un silicon compiler plecînd de la programe scrise în OCCAM (Martin 1986).

6.6 Repartizarea sistemului

Pare ironic că problemele întîlnite în proiectele de sisteme VLSI par să reflecte pe cele din societate. Capsulele evoluează rapid într-o societate cu două clase — în sistemele cu multe capsule VLSI există privilegiaţi care pot comunica local sau în interiorul capsulei şi nevoiaşi, care folosesc comunicaţii lente, în exteriorul capsulei. Datorită acestor inegalităţi repartizarea unui sistem devine foarte importantă. Cînd se distribuie un sistem pe circuite integrate, există trei considerente importante. Acestea sînt structura fiecărui circuit, pinii capsulei şi puterea disipată. De asemenea, problemele privind discrepanţa dintre performanţă în interiorul şi exteriorul capsulei vor influenţa modul de repartizare. Punctele de comunicaţie maximă trebuie menţinute în interiorul capsulei.

Este interesant de observat că circuitele celor mai multe firme producătoare păstrează punctul de comunicaţie maximă în exteriorul capsulelor, crescînd complexitatea sistemului tocmai prin încercările de a minimiza lărgimea de bandă, care poate provoca strangulări. Evident este vorba de interfaţa procesor-memorie, prin intermediul căreia se transmit codul şi datele în toate sistemele cu microprocesor. Soluţiile complexe includ memorii cache în interiorul capsulelor, pre-încărcarea instrucţiunilor şi execuţia lor pipeline, sau seturi de instrucţiuni mai complexe. De asemenea, pot exista excepţii, care implică integrarea funcţiilor de memorare şi prelucrare pe aceeaşi capsulă, o încercare deliberată de a reduce complexitatea care nu este necesară. Un exemplu bun în acest sens îl oferă transputerul INMOS, descris în § 3.5.5.

6.6.1 Structura

Idealul este de a integra întregul sistem pe o singură capsulă, deoarece altfel comunicaţiile externe sînt lente, consumă multă energie, fiabilitatea este inerent mai mică şi ocupă un volum mare. Totuşi, în cursul prelucrării cristalului de siliciu se produc defecte care pot determina o funcţionare defectuoasă a circuitelor. Aceste defecte includ:

(a) defecte în cristal;
(b) defecte ale măştilor folosite;
(c) defecte introduse în timpul prelucrării (de exemplu, particule străine);
(d) defecte introduse în cursul manipulărilor;
(e) defecte introduse de prelucrările necorespunzătoare (de exemplu, subţierea metalului);
(f) găuri mici între straturi;
(g) defecte produse cristalului în cursul prelucrării.

Datorită prezenţei acestor defecte, numai o parte dintr-un cristal (wafer) prelucrat va fi complet funcţional (presupunînd o proiectare corectă). De exemplu, să considerăm un cristal de 3 inch cu 100 capsule fabricate într-o tehnologie MOS. O structură tipică pentru astfel de cristale ar fi de aproximativ 30%, implicînd o medie de 30 capsule funcţional corecte pentru un cristal. De obicei se presupune că defectele ce intervin sînt distribuite aleatoriu şi cuantificate ca număr pe cm^2. În cursul ultimilor 20 de ani s-au obţinut progrese importante în tehnologia camerelor pure, a echipamentului folosit, materialelor şi măştilor. În consecinţă, se pot produce acum structuri rezonabile care pot ocupa chiar peste 1 cm^2.

Modelul cel mai simplu presupune o distribuire aleatoare a defectelor şi, mai mult, se consideră că un singur defect aflat oriunde va determina defectarea capsulei. În acest caz, probabilitatea de a găsi defecte pe o anumită capsulă se poate calcula cu distribuţia Poisson pe baza parametrilor D (densitatea defectelor) şi A (aria capsulei). Folosind statistica Poisson, probabilitatea ca o capsulă să fie bună este

$$P(D, A) = \exp(-DA)$$

Acest model nu reflectă cu acurateţe comportarea reală a procesului de fabricaţie, deşi reprezintă o aproximare bună a regiunii corespunzătoare structurilor foarte mici. Capsule cu o suprafaţă de mai multe ordine 1/D, vor avea o probabilitate extrem de mică de a nu conţine nici un defect. Pentru a produce structuri rezonabile, trebuie ca suprafaţa să fie menţinută la cîteva ordine 1/D. Pentru un proces bun, densitatea defectelor variază între 1 şi 5 cm^{-2}. Astfel, capsule de aproximativ 1 cm^2 vor asigura structuri modeste.

În realitate defectele nu sînt distribuite aleator, găsindu-se mult mai multe defecte la periferia cristalului decît spre centru. De asemenea presupunerea că defectele pot fi modelate ca puncte nu este corectă, deoarece multe defecte sînt mari, în comparaţie cu dimensiunile capsulei. Acestea se numesc defecte de suprafaţă. Matematic, este mult mai uşor de lucrat cu puncte decît cu suprafeţe. Defectele de suprafaţă pot fi modelate prin modificări ale funcţiei Poisson, considerînd că suprafaţa conţine un număr de puncte defecte. Prin urmare acest efect este modelat ca un ciorchine de defecte, iar distribuţia nu mai poate fi considerată aleatoare. Sistemul poate fi modelat printr-o superpoziţie de mai multe distribuţii aleatoare. Punctul slab al modelului Poisson simplu este că defectele sînt distribuite în urma mai multor etape de prelucrare, unele mai critice decît altele. De exemplu, un defect de mască la interfaţa metal 1 — metal 2 nu va fi fatală, dacă nu este inclusă în joncţiunea celor două metale. Evident, nu toate defectele vor provoca nefuncţionarea capsulei.

Deşi s-ar putea ca eliminarea completă a defectelor să fie imposibilă, se pot proiecta capsule care să funcţioneze în prezenţa unor defecte. Aceste tehnici, tratate în § 6.7, permit fabricarea economică a unor capsule mai mari. Aceste tehnici s-au folosit cu succes la proiectarea memoriilor. Structura regulată a memoriilor furnizează scheme deosebit de eficiente pentru evitarea defectelor introduse de procesul de producţie. Bineînţeles, ţinta finală este de a ajunge în situaţia de a folosi un cristal pentru un

întreg sistem. La nivelele curente de complexitate, ar fi vorba de sisteme conţinînd sute de milioane de porţi. În astfel de sisteme necesitatea multiplicării este evidentă.

6.6.2 Pinii capsulei

O altă limitare a repartizării sistemelor este numărul de pini ai unui circuit integrat. Această limită este impusă de restricţiile privind puterea disipată şi realizarea capsulei, şi contribuie la rîndul ei la limitarea posibilităţilor de comunicaţie cu exteriorul.

Prin urmare, o proiectare bună a unui circuit VLSI constă într-un sistem care îşi este într-o mare măsură suficient şi are cît mai puţine conexiuni la lumea externă. Totuşi, în acest mod se intră în conflict cu alte condiţii, de proiectare, pentru circuitele VLSI, ca acelea definite de partiţionarea structurilor conectate în mod regulat, ca masivele de procesoare. Problema este că odată cu creşterea numărului de procesoare implementate, lărgimea benzii de comunicaţie între capsule trebuie să crească. Această creştere poate varia fie cu suprafaţa procesoarelor incluse, ca în cazul conexiunilor la memorie externă aflate la capsulele SIMD, sau cu perimetrul masivului.

Conexiunile externe ale unei capsule sînt costisitoare; sudurile metalice din interiorul capsulei consumă o suprafaţă relativ mare, iar dimensiunea unei suduri (100—150 μm^2) trebuie să rămînă constantă, indiferent de modificarea dimensiunilor circuitului. Şi circuitele driver trebuie să-şi păstreze dimensiunea. Sudurile consumă o cantitate mare de energie, ceea ce poate influenţa considerabil liniile de alimentare, dacă mai multe suduri îşi schimbă starea în acelaşi timp (ca la o magistrală, de exemplu). Capsulele mari sînt costisitoare, şi ocupă o suprafaţă mare din placa logică. Deşi, tehnologiile de încapsulare asigură acum conexiuni de suprafaţă mică, densitatea pinilor face plăcile logice mai scumpe datorită straturilor suplimentare necesare pentru scăderea densităţii traseelor electrice.

Capsula cu un singur circuit nu este singura limită ce intervine la partiţionarea sistemului, deoarece în aceeaşi capsulă pot fi introduse mai multe circuite. Producătorii experimentează capsule care conţin hibrizi pentru uzul comercial. Ele au fost folosite în aplicaţii militare. Exemple sînt stratul gros de ceramică şi siliciul metalizat. Deşi aceste tehnici sînt relativ noi şi încă costisitoare, în comparaţie cu tehnologiile de încapsulare convenţionale, oricum avantajele lor evidente vor determina dezvoltarea lor în continuare.

Pentru realizarea unei repartizări bune a sistemului poate fi folosită teoria grafurilor. La un nivel de descriere dat, orice sistem poate fi definit cu un graf orientat, unde componentele sistemului sînt nodurile (porţi sau blocuri funcţionale), iar conexiunile reprezintă arcele grafului. O partiţionare bună împarte graful în subgrafuri, unde fiecare subgraf conţine un grad înalt de conectivitate, iar graful format prin partiţionare are un grad scăzut de conectivitate.

Situaţia nu este chiar atît de simplă, deoarece pentru nici un sistem nu există o singură implementare şi există întotdeauna un număr de compromisuri care se pot realiza pentru reducerea numărului de pini. De exemplu, semnalele pot fi codificate şi decodificate în interiorul capsulei, înaintea utilizării lor, la costul unui timp de întîrziere suplimentar. De asemenea, se poate folosi un singur pin pentru multiplexarea în timp a mai multor semnale.

Legătura INMOS implementată de transputer reprezintă un exemplu bun pentru un astfel de compromis. Transputerul are 4 legături de conectare la alte transputere, implementate ca o pereche de fire ce transmit un bait de date într-un pachet de 11 biţi. Legătura este bidirecţională şi poate transmite date şi recepţiona pachete în orice ordine. O mare suprafaţă din circuit a fost folosită pentru optimizarea vitezei de transmisie (20 MHz). Acesta este un bun compromis, deoarece o implementare paralelă ar necesita cel puţin 10 fire pentru o legătură. Scalînd aceste cifre cu un factor de 8, pentru a asigura cele patru legături bidirecţionale se realizează o diferenţă mare în numărul de pini. În acest mod, s-a micşorat viteza, în schimbul unei reduceri masive a numărului de pini şi a suprafeţei utilizate, care nu este însă o resursă critică.

6.6.3 Disiparea energiei

Ultima restricţie privind repartizarea sistemului se referă la puterea disipată. Există o limită de 2 W pe capsulă, în cazul tehnologiilor de încapsulare clasice, care folosesc plăci cu trasee imprimate şi răcirea forţată cu aer. Sînt necesare tehnologii de încapsulare mai sofisticate, care folosesc materiale ceramice, tehnologie de imersiune în lichid etc. Toate acestea ridică însă costul sistemului.

Tehnologia CMOS, care lucrează la temperaturi şi tensiuni scăzute este foarte rapidă, comparabilă acum cu ECL. Firma ETA construieşte calculatoare cu masive de porţi VLSI CMOS, ce operează la temperatura hidrogenului lichid (77°K) şi tensiuni scăzute. Se obţin viteze de 100—200 MHz.

La orice sistem există o componentă de putere consumată dinamică (P_d), datorită încărcării capacitive şi care este proporţionala cu numărul porţilor (N_g), frecvenţa medie de lucru $\langle f \rangle$ şi tensiunea la care comută (V_{DD}):

$$P_d = N_g \langle f \rangle \; C V_{DD}^2$$

unde C este capacitatea sarcinii.

Acesta este consumul important dintr-un circuit CMOS. În cazul tehnologiei NMOS mai există o componentă corespunzătoare tuturor porţilor în starea low. Ea este definită de

$$P_d = N_g \langle V_{dd} - V_{out} \rangle / R_{dep}$$

O valoare tipică pentru R_{dep} a unui circuit NMOS este de aproximativ 50 kΩ, de unde se obţine un consum de putere de 0,1 mW pentru o singură poartă.

6.6.4 Tehnici pentru reducerea consumului de putere

Există o serie de tehnici care pot fi folosite pentru reducerea consumului de putere a unui circuit integrat. În general acestea se pot aplica mai ușor tehnologiilor cu pierdere de sarcină, cum este NMOS.

(i) Prima tehnică constă în reducerea numărului de pini. Sudurile de conectare consumă o putere cu cîteva ordine de mărime mai mare decît porțile logice. Dacă numărul lor este mare, acest consum poate reprezenta un procent important din consumul total.

(2) Sudurile pot fi proiectate astfel încît vîrfurile de putere care apar la tranziții să fie minimizate.

(3) O altă tehnică constă în folosirea circuitelor logice dinamice, care nu posedă componenta de putere statică. În acest mod se reduce spațiul folosit de sistemele CMOS, cu o reducere mică a consumului de putere. La NMOS consumul de putere se reduce însă semnificativ.

(4) Există tehnici pentru reducerea pierderilor statice în sistemele NMOS, ca masivele de comutare cu tranzistoare, care nu consumă putere statică. Ele se pot folosi și în tehnologia CMOS, deși se preferă procesele n, deoarece dispozitivele cu canale n sînt cele „bune", avînd o modalitate mai mare.

6.7 Integrarea pe un wafer

Integrarea pe un wafer (WSI) constă în utilizarea unui întreg wafer procesat de siliciu pentru un singur sistem. Multe din dezavantajele remarcate la scalarea tehnologiilor VLSI vor dispare (altele vor rămîne). Oricum pentru a folosi un circuit cu anumite defecte, trebuie adoptate tehnici speciale.

Întotdeauna producătorii de circuite integrate au dorit de a plasa cît mai multe circuite pe o singură capsulă. Cele două sau trei ordine de mărime de creștere a complexității realizate în cursul ultimei decade în industria semiconductorilor (Augarten 1983) s-au obținut prin micșorarea dimensiunilor circuitelor. Dimensiunile capsulelor au rămas mai mult sau mai puțin constante, la ceva sub 1 cm². Liderii industriali, și în principal producătorii de memorii, au folosit tehnici tolerante la defecte și cu redundanță pentru creșterea structurii implementate pe capsule mai mari. Prin utilizarea extensivă a redundanței și relaxarea regulilor de proiectare în punctele critice, suprafața circuitului poate fi mărită, pînă la ocuparea întregii suprafețe a unui wafer. Prima încercare de a realiza un astfel de circuit s-a făcut acum 20 de ani. În Marea Britanie Sinclair Research a lansat un proiect pentru studiul posibilității de realizare a circuitelor integrate pe un wafer. Acest proiect este continuat de compania Anamartic. Primele produse vor fi dispozitive de memorie de masă solid state (Pountain 1986b). Produsele ulterioare vor avea capabilități de procesare la nivelul memoriei; acestea vor fi calculatoare paralele.

Avantajele potențiale ale circuitelor integrate pe un wafer sînt:

(a) viteza mai mare datorită conexiunilor mai scurte și a încărcărilor mai mici;

(b) consum mai mic de putere datorită numărului mai mic de suduri;
(c) densitate mai mare a interconectărilor posibile;
(d) volum mai mic;
(e) fiabilitate mai mare datorită numărului mai mic de conexiuni mecanice;
(f) costuri mai mici ale sistemului.

Împotriva acestora pot fi menționate următoarele dezavantaje:
(a) statistica asociată structurii realizate pledează împotriva unor circuite foarte mari;
(b) densități mai mari de putere;
(c) dificultăți de testare;
(d) costurile configurației (poate fi necesară o procesare specială);
(e) prototipuri mai lente și mai costisitoare;
(f) probleme la combinarea tehnologiilor;
(g) inadecvată pentru toate sistemele.

Multe din dificultățile potențiale ridicate de integrarea pe un wafer pot fi ameliorate prin alegerea corectă a arhitecturii. Evident, acestea trebuie să fie regulate și multiplicate.

6.7.1 Arhitectura pentru WSI

Arhitecturile adecvate pentru WSI sînt regulate, conțin cîteva module cu un grad înalt de multiplicare și conțin, în mod ideal, numai comunicații regulate locale. În acest mod se minimizează efortul de proiectare, un considerent major, iar efectele adverse ale micșorării lungimii conexiunilor sînt minimizate. Cîteva arhitecturi bune pentru WSI sînt memoriile, masivele sistolice și masivele de procesoare. Dar, cît de bune sînt acestea pentru prelucrarea datelor?

(*i*) *Memoria*

Memoria este o structură ideală pentru implementarea WSI. Este foarte regulată, constă din foarte puține module multiplicate (la memoria RAM dinamică, un tranzistor și un condensator și are interconexiuni regulate, linii pentru bit și cuvînt. De aceea, nu este surprinzător că sînt produsele cele mai solicitate care exploatează redundanța. Deși s-au întreprins mai multe experimente pentru realizarea circuitelor pe un wafer, numai micșorarea suprafeței structurii a permis ca circuitele de memorie să aibă un succes comercial. Totuși, acestea nu realizează prelucrări, iar disponibilitatea circuitelor de memorie a perpetuat arhitectura von Neumann, cu unitățile ei separate de calcul, de control și memorie

(*ii*) *Masive sistolice*

Masivele sistolice sînt masive regulate de mașini simple cu un număr finit de stări (FSMS), unde starea unei mașini la un moment dat este comună întregului masiv. Denumirea provine de la termenul medical „sistolă", folosit la descrierea inimii. Un algoritm sistolic prelucrează date ce ajung la celule pe direcții diferite la intervale regulate de timp. Pot interveni și

alte stări decît cele pentru controlul local al mașinii. Algoritmul ține cont de poziția datelor și propagarea lor prin masiv. Deși aceste mașini au o structură regulată și conțin celule simple, ele nu execută calcule cu un grad prea mare de generalitate, sau cel puțin nu atît timp cît mașina cu stări finite nu devine o mașină cu program memorat. cînd se obține un masiv de procesoare. Pentru mai multe informații vezi lucrarea lui Moore et al (1987).

(iii) Masive de procesoare

Cînd elementul care se multiplică este un calculator, și nu o mașină cu un număr finit de stări, structura poate fi numită masiv de procesoare. În sistemele de calcul cu multiplicare, cele mai importante aspecte sînt legate de scalare (vezi capitolul 3). În timp ce un procesor care conține numere mici poate comunica eficient prin magistrale ale sistemului sau memoria comună, în sistemele care folosesc numere mai mari, secvențialitatea acestor metode produce strangulări. Apoi, proiectantul este obligat să considere sisteme unde datele sînt transmise între procesoare. Comutarea datelor are loc fie în cadrul unei rețele cu topologie fixă, unde distanțele cresc odată cu numărul procesoarelor implicate, fie într-o rețea programabilă, unde costurile comutatorului respectă o lege pătratică. În siliciu, natura planară a mediului de conectare tinde să favorizeze o rețea planară.

6.7.2 O trecere în revistă a tehnicilor

Tehnicile folosite pentru creșterea suprafeței circuitelor cu o structură adecvată se împart în două categorii — tehnicile tolerante la defecte și cele cu redundanță.

(i) Circuite tolerante la defecte

Se pot proiecta circuitele pentru a tolera defecte, de obicei prin includerea unor operatori redundanți în modul de lucru al circuitelor. În acest mod se pot detecta defecte sau combinații defectuoase, care sînt mascate în cursul operării normale a circuitului. Aceste tehnici sînt de obicei costisitoare în termeni de suprafață și nu sînt adecvate pentru ansamblul wafer-ului. Ele pot fi folosite pentru asigurarea integrității circuitelor critice prin adoptarea altor scheme. Un domeniu unde aceste tehnici sînt foarte indicate este cel al circuitelor de memorie, unde se pot folosi operatori redundanți pentru detectarea și corectarea unui singur bit eronat din orice cuvînt (unitate de protecție) de memorie. Overhead-ul pentru aceasta este $\log_2 n)/n$, 50 % pentru date de 8 biți.

(ii) Redundanța modulară

Redundanța modulară constă din duplicarea, triplarea sau multiplicarea într-alt mod a unui circuit. Cu două circuite, dacă ele concordă la un set de teste, putem avea încredere că funcționează corect . Structura

circuitului analizat a fost micşorată. Cu circuite triple, dacă două dintre ele concordă, rezultatele furnizate de ele pot fi considerate corecte. Pentru realizarea acestui lucru este necesar un circuit de votare (voter). Un circuit de votare simplu nu va detecta, totuşi, o gamă largă de moduri de lucru defectuoase. Soluţia o oferă redundanţa triplă şi circuitele triple de votare. Această tehnică impune un overhead masiv, care poate fi inacceptabil. Se foloseşte în mod obişnuit în situaţiile unde este necesară operarea continuă a unui sistem, cînd se poate scoate din funcţionare şi înlocui modulul defect.

(iii) Alte scheme cu redundanţă

Alte scheme sînt avantajoase numai cînd circuitul are un grad înalt de regularitate. Acestea sînt schemele cele mai indicate ce pot fi folosite de toate arhitecturile regulate, ca cele de memorie sau anumite masive de procesoare. Datorită structurii regulate, se pot implementa circuite suplimentare, care pot fi incluse în sistem în cazul defectării unui modul. Un exemplu ar fi utilizarea liniilor şi/sau coloanelor de procesoare suplimentare într-un masiv dreptunghiular de procesoare. Cei mai mulţi producători de memorii folosesc această soluţie, care implică un overhead mic. Circuitele suplimentare pot fi incluse în sistem într-o manieră permanentă (etapă de fabricare după test), sau volatilă. În ultimul caz, sistemul poate fi „reparat" după detectarea defectelor „in-field".

În fig. 6.12 şi 6.13 se prezintă un exemplu de astfel de schemă de înlocuire, realizată în cadrul unor experimente la Universitatea Southampton (Bentley şi Jesshope 1986). Fig. 6.12 prezintă diagrama unui sfert de wafer, care este prezentat în fig. 6.13. Fiecare bloc de memorie este serial, iar ansamblul asigură un acces paralel, ca o memorie disc solid state. În fiecare sfert, una din opt coloane virtuale poate fi adresată şi ieşirile sale conectate la magistrala de ieşire. Fiecare celulă primeşte patru linii adiacente şi informaţie codificată de control care indică numărul celulelor defecte la stînga (0, 1, 2 sau 3 defecte). Utilizînd informaţia de control, se foloseşte linia decodificată corespunzătoare pentru a selecta celula. Deoa-

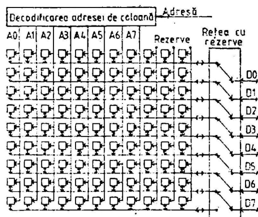


Fig. 6.12 Modul de operare al unui circuit de memorie integrat pe un wafer, cu redundanţa bi-dimensională

rece fiecare linie are propria magistrală de control, ea poate elimina oricare trei celule defecte. Dacă o linie conține mai mult de trei celule defecte, un semnal de detectare a erorii va indica rețelei suplimentare să ignore acea linie. În fiecare sfert se poate elimina numai o singură linie complet de-

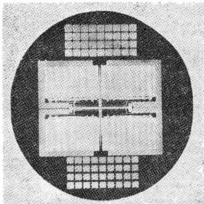


Fig. 6.13 O fotografie a circuitului prezentat în fig. 6.11 (diametru de aproximativ 7,5 cm).

fectă. În acest mod se poate configura fiecare sfert de memorie, cu oricare opt celule bune din unsprezece din fiecare linie și opt linii bune din nouă existente.

Wafer-ul se poate auto-configura, ca în modul test, prin scrierea unui cod în toate blocurile în paralel. Acest cod este recunoscut după testarea circuitelor și a magistralei, iar un circuit latch memorează starea bună sau rea, după terminarea testului. Acest latch de stare este folosit apoi pentru a genera semnalele de control pentru fiecare linie. Evident prin acest proces nu s-a testat latch-ul de stare și nici circuitele de control, care însă trebuie să poată fi verificate prin echipamente externe de test.

Acest wafer experimental nu are structura de comunicare corectă pentru un masiv de tip caroiaj. S-au propus, de aceea, modificări ale acestei scheme (Jesshope și Bentley 1986, 1987) ce urmează să se implementeze în siliciu. Aceste scheme vor permite fabricarea a 16×16 elemente procesoare RPA pe un wafer de 3 inch, cu tehnologie CMOS 3μm. Reguli de proiectare mai agresive (să spunem 1μm), și un wafer mai mare (să spunem de 5 inch), vor permite o creștere de 25 ori a densității estimate mai sus.

S-ar atinge astfel 200 Mflop/s pentru operaţii în virgulă mobilă şi pînă la 2 Gflop/s pentru operaţii cu întregi pe 16 biţi — o performanţă impresionantă pentru un calculator care va încăpea în buzunar.

Nu putem trata acest subiect complet aici, cei interesaţi avînd posibilitatea să consulte lucrarea editată de Jesshope şi Moore (1986). Aceasta reprezintă lucrările unei conferinţe ce a avut loc la Universitatea Southampton, unde au fost prezentate trei proiecte de wafer; cel produs de Sinclair Research se autoconfigura. Ca şi cel proiectat la Southampton, şi acesta era un circuit de memorie, configurat dintr-un lanţ liniar de celule.

6.8 Ultimul cuvînt (Încheiere)

Cu tehnologia care realizează progrese ce par a nu avea limite, nu este nici o îndoială că, cu toate problemele lor, calculatoarele paralele se vor impune tot mai mult. Din păcate tehnologia software nu a ţinut pasul cu progresele rapide realizate în domeniul hardware. Motivul pare a fi reprezentat de forţele ce guvernează aceste dezvoltări; setea insaţiabilă de mai multă putere de calcul. Aceasta a condus la respingerea realizărilor mai abstracte ale tehnologiilor software, în favoarea sistemelor înalt optimizate, mai mature. Oricum, utilizarea calculatoarelor paralele în domenii unde sistemele sînt mult mai complexe, a determinat dezvoltarea unor metodologii de programare mai abstracte (Harland 1984, 1986). Se cere o creştere a nivelului de abstractizare al hard-ului maşinii, odată cu exploatarea naturii tehnologiei, a posibilităţilor şi limitelor ei. Sarcina nu este uşoară, deoarece ambii membri ai ecuaţiei se modifică rapid. Convergenţa acestor discipline, în cadrul tendinţei actuale în învăţămîntul superior este un lucru de care va trebui să se ţină seama în viitor.

Postfață*

În urmă cu peste două milenii Heraclit afirma că: „nimic nu este de durată cu excepția schimbării", iar la începutul anilor 1970, în lucrarea „Șocul viitorului", Alvin Toffler sublinia faptul că „schimbările dramatice au devenit în esență procese stabile". Se poate aprecia că domeniul calculatoarelor este unul dintre domeniile în care aceste observații se manifestă cu cea mai mare pregnanță și în maniera cea mai benefică.

Începînd cu mijlocul deceniului șase, calculatorul modern evoluează în sensul satisfacerii cerințelor de calcul în continuă creștere, a numeroase activități social-economice. S-au dezvoltat clase diferite de calculatoare, pînă la calculatoarele de astăzi larg răspîndite, care acoperă un spectru de dimensiuni și aplicații mai mare de cît al oricărui alt produs manufacturat. Costul unor asemenea obiecte manufacturate, care se pot numi calculatoare, se plasează într-o gamă cuprinsă între 20 dolari, pentru dispozitivele programabile de buzunar, și 20 milioane de dolari, pentru cele mai mari sisteme de tip supercalculator, aflate pe piață în prezent. Acest factor, de peste șase ordine de mărime, se referă la cost, putere de calcul, capacitate de stocare a informației etc. Pentru satisfacerea marii diversități a aplicațiilor calculatoarelor, cît și a posibilităților financiare ale cumpărătorilor potențiali, producătorii de calculatoare continuă să dezvolte, într-un ritm rapid, noi tipuri de calculatoare și software aplicativ.

Deși există o multitudine de termeni în uz, în prezent, pentru a identifica clasele de calculatoare, se pare că asupra acestor clasificări nu există un consens și că nu se manifestă nici precizia necesară. În continuare se încearcă o clasificare generală a sistemelor de calcul avînd în vedere evoluția și aplicațiile acestora.

La începutul erei calculatoarelor moderne, în anii 1950, se considera că toate mașinile sînt la fel de utile pentru toate genurile de aplicații. Gama aplicațiilor era destul de diversă, dar măsura în care se foloseau calculatoarele era relativ limitată. Calculatoarele foloseau reprezentarea numerică binară sau zecimală. Rapid s-au dezvoltat două piețe: una pentru aplicațiile economico-financiare, iar alta pentru aplicațiile științifice. Pentru aplicațiile economico-financiare s-a considerat potrivită reprezentarea zecimală, în virgulă fixă, iar pentru aplicațiile științifice — reprezentarea binară, în virgulă mobilă.

La sfîrșitul deceniului șase și începutul deceniului șapte, cele două piețe s-au dezvoltat mai mult sau mai puțin separat, avînd fiecare o cultură specifică. În mod obișnuit arhitecturile calculatoarelor s-au concentrat pe

* Prof. dr. ing. A. Petrescu, Institutul Politehnic București

una sau alta din aceste piețe. Incepînd cu anul 1964, care a marcat apariția sistemelor IBM-360, producătorii de calculatoare au încercat să realizeze cel puțin hardware-ul capabil să satisfacă ambele piețe. Astfel, arhitectura calculatoarelor medii-mari, dezvoltate la mijlocul deceniului șapte, includea atît aritmetica zecimală, cît și pe cea în virgulă mobilă. S-au implementat sisteme de operare unice, iar pentru a satisface necesitățile fiecărei piețe, au fost dezvoltate limbaje de nivel înalt și programe de aplicații specifice. În cursul anilor 1970—1980 aceste mașini s-au implementat ca mașini de calcul universale, capabile să satisfacă ambele clase de aplicații. În prezent ele sînt cunoscute sub numele de „mainframes".

La începutul deceniului șapte se constată necesități pregnante legate de existența unor calculatoare pentru aplicații științifice mult mai puternice decît calculatoarele universale (mainframes) existente în acel moment. La capătul superior al spectrului calculatoarelor științifice s-a dezvoltat un nou segment de piață datorită necesităților de apărare și datorită necesităților manifestate de comunitățile științifice implicate în studiile meteorologice și seismice. Calculatoarele care servesc acest segment de piață poartă numele de supercalculatoare. Ele reprezintă cele mai puternice calculatoare existente la un momenet dat, pentru calculele științifice. Conform definiției date în „Supercomputing, An informal Glossary of Terms", elaborat de „Scientific Supercomupting Subcommittee", de pe lîngă „IEEE Committee on Communications and Information Policy": „*Supercalculatorul/Supercalculatoarele reprezintă, la un moment dat acea clasă de calculatoare universale, care sînt mult mai rapide decît competitoarele lor destinate calculelor economice-financiare și au suficientă memorie centrală pentru a stoca informația necesară rezolvării problemelor pentru care sînt proiectate. Memoria calculatorului, productivitatea, viteza de calcul, cît și alte capabilități de acest gen contribuie la obținerea performanțelor dorite. Astfel, se constată că nu există o măsură cantitativă a puterii calculatorului pentru prelucrările științifice de mare anvergură. In acest sens este dificil de formulat o definiție precisă a supercalculatoarelor*".

Necesitățile privind calculatoarele rapide, destinate calculelor științifice, persistă de peste patru decenii. La jumătatea deceniului opt, capabilitățile supercalculatoarelor au depășit pragul critic, ceea ce a permis dezvoltarea explozivă a „*Științei Computaționale*", care, în prezent, se constituie de facto, alături de științele teoretice și experimentale, ca un domeniu legitim, de sine stătător.

La scurt timp după apariția calculatoarelor moderne, comunitățile științifice/tehnice și comercial/financiare au ajuns în situația de a utiliza calculatoarele într-o manieră susținută . Sistemele existente în acea perioadă erau voluminoase și costisitoare. Costurile ridicate au împiedicat proliferarea sistemelor mari — universale de calcul, în ciuda unei piețe deosebit de mari, aflată în așteptarea unor calculatoare mai puțin costisitoare. O serie de companii (DEC, IBM) au lansat pe piață sisteme de tip minicalculator, avînd viteze de calcul modeste dar și costuri mult reduse proporțional. În acest sens se pot aminti calculatoarele PDP-1 și IBM-650. Alte companii le-au urmat rapid exemplul. *Astfel, pe la mijlocul deceniului șapte*, deși piețele calculatoarelor destinate aplicațiilor științifice și comerciale erau distincte, *se puteau achiziționa trei tipuri de*

echipamente: *minicalculatoare*, *calculatoare universale "mainframes" și supercalculatoare*. Ca în cazul tuturor noilor produse, au fost realizate numeroase variante, capabile să acopere necesitățile diferiților beneficiari. S-au păstrat categoriile de bază, dar au apărut și noi sisteme caracterizate prin diferențe în ceea ce privește: performanțele, dimensiunea și tipul memoriei, lungimea cuvîntului și tipul instrucțiunilor. În continuare a fost diversificat software-ul pentru a satisface necesitățile unei mari varietăți de utilizatori. Factorii majori care au determinat succesul diferitelor oferte se refereau la preț, dimensiuni, ușurința de utilizare, performanțele relative etc. Prețul a jucat un rol mai important decît îl joacă în prezent. De exemplu, minicalculatoarele au asigurat un segment de piață, dar costul și performanțele lor variază încă foarte mult, plecînd de la minicalculatoarele relativ mici și ajungînd la capătul superior al gamei, la superminicalculatoare, care sînt mult mai puternice decît multe „mainframes" aflate la capătul inferior al acestei clase.

Deceniul opt a marcat apariția a două caracteristici arhitecturale pentru supercalculatoare: prelucrarea de vectori și prelucrarea paralelă. Acestea au avut ulterior repercursiuni apreciabile asupra noilor sisteme de calcul. Caracteristicile menționate au fost mai întîi implementate în sistemele: CDC STAR-100, avînd posibilitatea de a prelucra vectori, și în Illiac IV, procesor paralel, proiectat la Universitatea din Illinois și construit de firma Burroughs. Dintre cele două abordări, prelucrarea de vectori a fost introdusă mai întîi în supercalculatoarele comerciale, asigurînd o importantă creștere a performanțelor. Se apreciază că perfecționarea, în continuare a acestor tipuri de supercalculatoare, folosind tehnici ce se referă la arhitectură, tehnologii și software, este pe cale de a atinge saturația. Tehnologiile folosite în generațiile succesive de supercalculatoare ating treptat limitele naturale impuse de viteza finită a luminii și de materialele folosite. În consecință, sporirea majoră a performanțelor rezulta din proiectarea unor arhitecturi capabile să execte prelucrări paralele, în cadrul unui singur calcul/algoritm. Toți producătorii majori de calculatoare introduc acest concept în liniile lor de producție.

Costurile tipice ale supercalculatoarelor sînt de ordinul a 10—20 milioane de dolari. Cu toată scăderea spectaculoasă a raportului cost/performanțe, costul ridicat al investiției într-un supercalculator limitează larga răspîndire a acestora. Pentru a facilita accesul la supercalculatoare al unor largi grupuri de utilizatori, s-a recurs la interconectarea acestora prin linii de transmisii rapide de date. Supercalculatoarele au fost prevăzute cu calculatoare de comunicații specializate (front end computers), care au preluat toate sarcinile privind gestiunea transmisiilor de date. Datorită costului ridicat al supercalculatoarelor, cît și complicațiilor suplimentare legate de teleprelucrare, s-au căutat soluții alternative.

În deceniul nouă aceste căutări s-au concretizat prin apariția *minisupercalculatoarelor*. Acestea sînt relativ ieftine, ușor de procurat, au performanțe ridicate, fiind similare supercalculatoarelor sub aspect arhitectural. Datorită costurilor reduse ele sînt accesibile și unor grupuri restrînse de utilizatori, de tipul departamentelor din cadrul unor companii sau universități. În proiectarea minisupercalculatoarelor, o atenție deo-

seb ită se acordă în primul rînd costurilor şi în al doilea rînd performanţelor. Astfel, se susţine ideea „supercalculatorului personal", tot mai des întîlnită printre cercetători, care urmăresc să aibă controlul local al resurselor.

În dezvoltarea minisupercalculatoarelor un impact deosebit l-au avut *microprocesorul* şi *prelucrarea paralelă*. Încorporarea în minisupercalculatoare a unor microprocesoare performante, în cadrul unor arhitecturi paralele, a permis apariţia unui adevărat segment de piaţă pentru aceste sisteme, care sînt folosite de-sine-stătător (stand alone), încorporate în structuri distribuite, la nivel departamental, sau în calitate de calculatoare frontale puternice.

Fenomenul *calculatoarelor personale* a căpătat o deosebită amploare în deceniul nouă, datorită costurilor scăzute, puterii superioare de calcul, fiabilităţii ridicate şi uşurinţei de exploatare a acestui tip de echipamente. Ele îşi găsesc utilizări în prelucrări de texte, în gestiunea bazelor de date, în tabelarea electronică etc., transformînd birourile în adevărate birouri electronice (electronic office).

Relativ recent a apărut pe piaţă o clasă foarte puternică de calculatoare personale, denumite iniţial *staţii de lucru ştiinţifice*. Costurile lor în funcţie de performanţe, variază de la cele ale unor calculatoare personale puternice pînă la cele ale minicalculatoarelor sau ale calculatoarelor universale — „mainframes" aflate la capătul inferior al gamei.

Evoluţia industriei de calculatoare poate fi urmărită examinînd: clasa aplicaţiilor, costurile sistemelor, cifra de afaceri în fiecare segment. În figura P 1 se prezintă modificările intervenite în spaţiul ce defineşte tehnica de calcul prin prisma indicatorilor menţionaţi mai sus. Axa aplicaţiilor are un caracter calitativ separînd aplicaţiile ştiinţifice de cele economico-financiare. Axa costurilor pe unitate indică costul diferitelor sisteme în mii de dolari. Axa verticală specifică cifra anuală de afaceri în dolari. Ultimele două axe folosesc scări logaritmice.

În 1960 (fig. P1 a) cifra totală de afaceri a fost de circa 700 milioane de dolari, existînd o separare distinctă între calculatoarele destinate calculelor ştiinţifice şi cele destinate calculelor economico-financiare.

Între 1965 şi 1975, piaţa minicalculatoarelor a preluat o importantă parte a totalului aplicaţiilor. În figura 1 b se observă cum sistemele universale — „mainframes" au acoperit pieţele aplicaţiilor ştiinţifice şi economico-financiare.

Perioada 1975—1980, avînd ca an de referinţă 1975 (fig. P1 c) marchează impunerea supercalculatoarelor ca şi, de altfel, a calculatoarelor personale.

În prima jumătate a deceniului nouă (fig. P1 d) se manifestă o puternică creştere a tuturor segmentelor, cît şi apariţia staţiilor de lucru, care se plasează între calculatoarele personale şi minicalculatoare.

Cea de-a doua jumătate a deceniului nouă, marcată prin anul 1987 (fig. P1 e), ilustrează creşterea masivă a cifrei de afaceri în toate segmentele pieţei de calculatoare, evidenţiind în acelaşi timp apariţia unui nou segment, acela al minisupercalculatoarelor.

În cadrul fiecărei etape au fost dezvoltate şi alte clase de calculatoare; acestea, însă nu au căpătat o pondere importantă pentru a se putea identifica un segment specific de piaţă. Un asemenea exemplu îl constituie procesoarele matriciale.

ANNUAL $
REVENUE
$1B
$0.1B
BUSINESS
COMPUTERS
($.371 B)
SCIENTIFIC
COMPUTERS
($.300 B)
1
10
100
1000
10000

(a) 1960

$10B
$1B
$0.1B
MINI
COMPUTERS
($.840 B)
MAINFRAMES
($4.44 B)
1
10
100
1000
10000

(b) 1970

$100B
$10B
$1B
$0.1B
PERSONAL
COMPUTERS
($.086 B)
MINI
COMPUTERS
($5.75 B)
MAINFRAMES
($9.92 B)
SUPERCOMPUTERS
($.024 B)
1
10
100
1000
10000

(c) 1977

ţiona faptul că un singur cadru recepţionat de satelitul LANDSAT conţine 30 milioane octeţi şi că pentru acoperirea unei zone geografice avînd suprafaţa Moldovei sînt necesare 13 cadre. Seturi de imagini ale globului pămîntesc se realizează la intervale de 15 zile. Pentru prelucrarea unei asemenea cantităţi uriaşe de informaţii, astfel acumulate, se impune folosirea unor resurse de calcul extrem de puternice.

Explorarea resurselor energetice a condus la numeroase studii privind explorările seismice, modelarea zăcămintelor de hidrocarburi, energetica nucleară s.a. Pentru prelucrarea celor 10^{15}—10^{17} biţi de date seismice, care se acumulează anual, cît şi pentru modelarea comportării plasmei în generatoarele Tokamak sînt necesare sisteme de calcul cu pînă la două ordine de mărime mai puternice decît cele existente.

Cercetările în domeniul medical se referă, printre altele, şi la utilizarea calculatoarelor în tomografie, inginerie genetică ş.a. Pentru implementarea tomografiei în timp real sînt necesare sisteme de calcul cu viteze de 2—3 Gflops. De asemenea, decodificarea codului genetic, evaluarea mutaţiilor genetice şi proiectarea de noi organisme impun folosirea unor calculatoare cu viteze foarte mari.

● *Utilizarea supercalculatoarelor creează o nouă dimensiune în domeniul cercetării ştiinţifice fundamentale şi aplicative, caracterizate, în principal, prin activităţi de natură teoretică şi experimentală.* În prezent *triada Teorie — Experiment — Calcul* asigură condiţiile dezvoltării fără precedent a activităţilor de cercetare ştiinţifică în cele mai multe domenii ale activităţii social economice.

Teoria sugerează noi experimente şi interpretează rezultatele experimentelor; în ceea ce priveşte calculul, teoria furnizează ecuaţii/modele matematice şi interpretează rezultatele.

Experimentul sugerează noi ipoteze teoretice şi testează teoria, de asemenea, generează date pentru calcul.

Calculul, pe baza unor operaţii precise, şi laborioase, sugerează noi modele teoretice; în legătură cu experimentul, calculul modelează procese fizice, sugerează noi experimente, analizează date şi controlează echipamente. Practic în fiecare disciplină ştiinţifică şi inginerească , dezvoltarea tehnologică se bazează din ce în ce mai mult pe simulare, cu ajutorul calculatorului, deoarece numai o mică parte a experimentelor dorite pot fi realizate fizic la costuri realiste. Această abordare, bazată pe simulare, a stimulat şi mai mult cererea de putere de calcul.

● *Supercalculatoare: clase, performanţe, costuri, exemple.*
După puterea de calcul şi după cost, supercalculatoarele se împart în trei categorii: supercalculatoare propriu-zise (full scale), supercalculatoare medii (near supers) şi minisupercalculatoare.

● *Supercalculatoarele* au performanţe cuprinse între 0.2—3 Gflops la costuri de ordinul a 2—25 milioane de dolari. Ca exemple se pot da: Cray-2, Cray X-MP, Cray Y-MP, NEC SX-1/SX-2/SX-3, Fujitsu/Amdahl VP200/VP400E, Fujitsu VP2000/VP2600/20, Hitachi S-820-80, CDC/ETA 10G, IBM GF11/3090S.

• *Supercalculatoarele medii* se caracterizează prin viteze de ordinul a 50÷500 Mflops și costuri de 1÷4 milioane de dolari. În această clasă sînt cuprinse : IBM 390/VF, Loral MPP, CDC Cyber-plus, Univac 1194/ISP, Connection machine, BBN butterfly etc.

• *Minisuprecalculatoarele* operează la viteze de 10—100 Mflops, avînd costuri de 0,1—0,5 milioane de dolari. Ca exemple se pot da sistemele : Alliant FX/8 , Convex C-1, SCS-40, Elxsi, 640, FPS 14 Max, T-series, Wrap etc.

Pentru comparație se prezintă și unele date referitoare la *superminicalculatoare*. Viteze : 0,5 Mflops sau 5 Mips. Cost : între 20 și 400 mii de dolari. Exemple : VAX 6800, VAX 11/780, IBM 4300, IBM 9370- Pyramid etc.

• *Evoluția și principiile constructive ale supercalculatoarelor*. Realizarea unor sisteme de calcul cu performanțe care se plasează în zona supercalculatoarelor a fost posibilă prin perfecționări de ordin tehnologic și funcțional-structural. Pe plan tehnologic supercalculatoarele au folosit în cele mai multe cazuri componente de mare viteză (circuite ECL, GaAs), și tehnici evoluate de împachetare-răcire, ceea ce a permis reducerea dramatică a perioadei ceasului sistemului (2,9 ns la NEC SX-X/SX-3/44) *Sub aspect funcțional/structural*, supercalculatoarele fac uz intens de paralelism la toate nivelurile de operare.

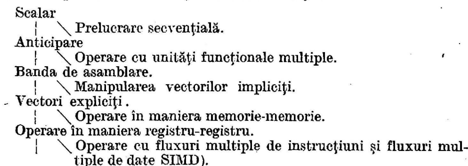
Astfel, *la nivelul unităților funcționale*, paralelismul se implementează sub forma prelucrării paralele a biților care reprezintă datele. Paralelismul la nivelul elementelor de prelucrare se materializează prin efectuarea diferitelor operații în paralel, cu operanzi multipli, și prin funcționarea în regim de *bandă de asamblare (pipeline)*.

În sistemele uniprocesor paralelismul se poate manifesta în cadrul accesului la memorie, anticipării instrucțiunilor, suprapunerii operațiilor de prelucrare cu operațiile de I/E.

Cele mai multe supercalculatoare reprezintă sisteme multiprocesor cu comunicații prin *partajarea memoriei* și prin *transfer de mesaje (memorie distribuită)*. În primul caz, data scrisă în memorie, de către un procesor, poate fi citită de către toate procesoarele din sistem,în cadrul unui sistem sincronizat de acces. În cazul al doilea, comunicația se face de la punct la punct. Data este generată de către procesorul sursă și furnizată la procesorul destinație. În cazul în care nu există o cale directă de comunicație, între sursă și destinație, mesajul este transferat prin procesoare intermediare. *Arhitecturile de interconectare pentru structurile cu memorie partajată se bazează pe magistrale și pe rețele de comutare cu mai multe etaje, incluzînd și comutatorul de tip „Crossbar". Arhitecturile de conectare pentru sistemele cu memorie partajată folosesc conexiuni de tip hypercub (rețea/plasă).*

Dezvoltarea operării concurente a fost abordată pe mai multe căi : banda de asamblare, utilizarea unităților funcționale multiple în sistemele uniprocesor, sisteme bazate pe procesoare care cooperează, sisteme specializate de calcul s.a.

Evoluția structurii sistemelor de calcul, de la prelucrarea secvențială, la prelucrarea paralelă-concurentă, este dată mai jos :



Operare cu un singur flux de instrucțiuni și cu fluxuri multiple de date (SIMD).

● În acest context *supercalulatoaree se pot clasifica după structurile operaționale și de interconetare,* după cum urmează.

— *Uniprocesoare cu unități funcționale multiple și cu opțiuni hardware pentru prelucrarea vectorilor.* Sisteme reprezentative : Alliant FX/1, IBM 3090, CDC 7600, FPS, 164/264/36, Convex, Cray 1, Cray X-MP/1, Cray 2, Cray Y-MP/1, Cyber 205, Amdahl 500/1100/1200/1400, Fujitsu VP-50/100/20/400/400 E, Hitachi S-810/820-80, NEC SX-2, SCS-40 etc.

— *Procesoare matriciale SIMD, procesoare atașate.* Sisteme reprezentative : Coral MPP, ICL/DAP, FPS 164/MAX, Connection Machine, IBM GF-11 ș.a.

— *Sisteme multiprocesor cu memorie partajată.* Sisteme reprezentative : Cray X-MP/2/4, Cray-2 2/4, Alliant FX/8, Encore/Multimax, Elxsi 6400, Sequent 8000, IBM 3090/400 VF, Unicac 1194/ISP etc.

— *Multicalculatoare cu memorie distribuită.* Sisteme reprezentative : IPSC, Amtek 14, NCUBE, BBN Butterfly, CDC Cyberbplus, Culler PSC, FPS, T-series, Wrap etc.

— *Sisteme ierarhice configurabile.* Sisteme reprezentative : Cedar, ETA-40, IBM RP3, Remps ș.a.

● Un aspect important, privind structura sistemelor de calcul paralel, se referă la *organizarea memoriei : partajată, distribuită.*

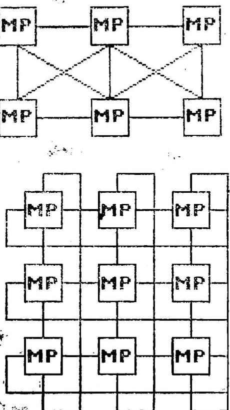
În cazul memoriei partajate procesoarele (P) și memoriile (M) se constituie în unități distincte, care se interconectează cu ajutorul comutatoarelor (S) „Crossbar", al comutatoarelor cu mai multe etaje sau folosind magistrala unică (fig. P2).

În cazul memoriei distribuite, fiecare procesor P are asociată o memorie M proprie, formînd astfel unități distincte MP, care se pot interconecta sub formă de rețea/plasă (fig. P3).

● *Tipuri reprezentative de supercalculatoare, calculatoare de foarte mare capacitate.*

În prezentarea care urmează (Tabelul P1) se vor folosi următoarele notații : MP-multiprocesor ; MC-multicalculator ; SM-memorie partajată ;

Fig. P2 Conexiuni MP cu memorie partajată. Comutator ,,CROSSBAR", comutator cu mai multe etaje; conexiune pe magistrală.



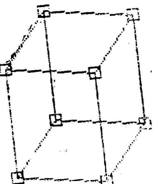


Fig. P3 Conexiuni MP — MP cu memorie distribuită. Interconectare de tip rețea/plasă; conexiune de tip hipercub.

Tabelul P1 Tipuri reprezentative de supercalculatoare de capacitate mare.

Modelul	Arhitectură/ Configuraţie	Nr. maxim de procesoare	Tip procesor tehnologie	Cap. max. a memoriei	Performanţe
Cray X-MP/4	MP cu SM şi conexiuni directe	4 Proc.	custom ECL	16 Mw in CM 128 Mw in SSD	840 Mflops
Cray 2	idem	4 Proc., 1 IOP	idem	256 Mw	2 Gflops
Cray 3	Mp cu SM	16 Proc.	custom GaAs/ECL	2 Gw	16 Gflops
Cyber-205	UP cu Proc. ptr. scalari şi 4Ba ptr. vectori	1 Proc.	custom CMOS	4 Mw	400 Mflops
ETA-10	MP cu SM	8 Proc. 16 IOP	custom	256 Mw	10 Glops
Fujtsu VP-200	UP cu BA multiple	1 Proc.	custom ECL	32 Mw	533 Mflops
Hitachi S-810	UP cu BA multiple	1 Proc.	custom	32 Mw	840 Mflops
HEP-1	MP cu SM şi reţea de comutare	16 Proc.	custom	256 Mw	160 Mflops
IBM 3090/400 VF	MP cu SM şi conexiuni directe	4 Proc.	custom CMOS	2 GB in CM 16 TB in EM	480 Mflops
CDC Cyberplus	MC cu DM şi conexiune in inel	64 Proc.	custom	512 Kw/procesor	65 Mflops, 620 Mips/procesor
Connection machine	SIMD cu DM hypercube in reţea globală	64 k. PE	VLSI/CMOS Gate array	32 MB	250 Mflops 1 Gips
BEN butterfly (fig. P4)	MD cu SM şi reţea de comutare buttefly	256 Proc.	M 68020 şi coprocesor custom	128 Mw	256 Mips
IBM GF11	SIMD cu reţea configurabilă Bencş	576 PE	cîstom Fl. P. Proc.	2 MB/proc. 1. 1GB-total	20 Mflops 11 Gflops total
IBM RP3	MP cu SM/ DM şi reţea de conectare	512 Proc.	RISC/32 biţi	128 Mw	800 Mflops 1300 Mips
Cedar	MP-ierarhic cu SM	256 Proc.	Alliant/Fx cluster	256 Mw	3.2 Gflops

DM-memorie distribuită; UP-uniprocesor; IP-procesor interactiv; IOP -procesor de I/E; PE-element de prelucrare; CM-memorie centrală; SSD-dispozitive tranzistorizate/integrate; BA-banda de asamblare; EM — memorie externă.

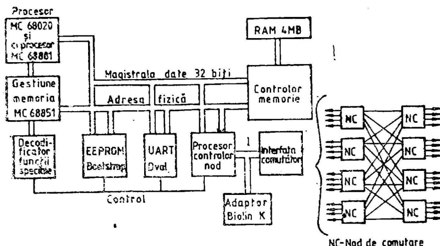


Fig. P4. Schema bloc pentru un procesor de nod ,,butterfly'' şi specificarea comutatorului ,,butterfly''.

În Tabelul P2 se prezintă date asupra unor supercalculatoare avînd benzi de asamblare, care operează cu cuvinte de 64 de biţi.

În anul 1983, Subcomitetul Ştiinţific IEEE, pentru Supercalculatoare a elaborat un raport privind dezvoltarea supercalculatoarelor în SUA şi în alte ţări. Faţă de concluziile acelui raport, în 1989 se constatau următoarele elemente:

— Control Data a renunţat la producţia de sisteme ETA;

— Din firma Cray Research s-a desprins un grup, care a format compania de supercalculatoare Supercomputer Systems, avînd ca sponsor firma IBM;

— În mai 1989, din firma Cray Corporation s-a desprins firma Cray Computer Corp., pentru a produce sistemele Cray-3.;

— IBM a reintrat în cursa pentru supercalculatoare, prin conectarea unor subsisteme pentru prelucrarea vectorilor la sistemul ,,mainframe'' 3090S. Acesta se situează în gama medie-mică a performanţelor supercalculatoarelor. Pînă la sfîrşitul anului 1988 au fost comercializate circa 300 sisteme;

— În aprilie 1989 NEC a anunţat un nou supercalculator de tip multiprocesor, acesta fiind primul model japonez de multiprocesor.

Se constată că japonezii au produs cele mai performante supercalculatoare de tip uniprocesor, în speranţa de a le conecta ulterior, în structuri paralele.

Tabelul P3 furnizează unele caracteristici ale celor mai recente supercalculatoare sau ale supercalculatoarelor în faza de prototip.

Tabel P2. **Caracteristicile unor super-**

	Cray X-MP	Cray-2	Cray Y-MP	CDC/ETA 10G
Anunțarea seriei	—	—	Feb. 1988	Oct. 1985
Prima instalare	Iunie 1983	Oct. 1984	Oct. 1988	1987
Număr procesoare	1, 2, 4	1, 2, 4	1, 2, 4, 8	1, 2, 4, 8
Performanțe max.				
Pe procesor.	235 Mflops	488 Mflops	333 Mflops	572 Mflops
Total vîrf.	940 Mflops	1951 Mflops	2670 Mflops	4600 Mflops
Durata ciclului	8,5 ns	4,1 ns	6 ns	7 ns
Cap. max. memorie	0.5 Gbyte	4 Gbyte	1 Gbyte	1 Gbyte
Extensie memorie	4 Gbyte	—	4 Gbyte	8 Gbyte
Tehnologie				
Logică	ECL 16 porti	ECL 16 porti	ECL 2500 porti	CMOS 20 K porti
Intîrzierea pe poartă	650 ps	650 ps	350 ps	300 ps
Memoria principală	CMOS 64 K	MOS 1 M DRAM	ECL	SRAM 256 K
Ciclul	68 ns ECL SRAM 64 K; 34 ns	185 ns SRAM 256 K; 54 ns	30 ns —	35 ns DRAM 1 M; 150 ns
Memorie I/E; disc; monolitică secundară				
Nr. căi (total)	32; 2	36; —	48; 4	16; —
Debit max. cale (Mbytes/s)	26,6; 1250	26.6; 500	26.6; 1250	50; —
Debit max. agregat (Mbytes/s)	40; 2500	2000; —	800; 5000	1000; —
Răcirea	Freon	Imersiune Fluorinert	Fluorinert UCP și Mem. Freon I/E	Azot (77 K) Logica Freon Mem.
Sistem operare	CTSS/COS/ Unicos	Unicos/CTSS	CTSS/COS/ Unicos	SYS V, EOS
Limbaje/compilatoare	Fortran 77, C, Pascal, Ada	Fortran 77, C, Pascal Ada	Fortran 77, C, Pascal, Ada	Fortran 77, C, Cybil, Meta asmb.
Caracteristici				
Benzi asamblare	4 Adunare 4 Logice 4 Deplasare 4 Contorexp. 4 Imulț. VM 4 Adun.	4 Inmulțire 4 Adunare 4 Logice 4 Intregi 4 Contorexp 4 Incarcă	8 Adunare 8 Logice 8 Deplasare 8 Contor exp. 8 Înmulț. VM 8 Adun VM	16 Adunare 16 Întregi 16 Împărt. 16 Înmulț 16 Deplas. 16 Logice
Încărcă/ Memorează	8 Încarcă și 4 Memorează	sau 4 Memorează	16 Încarcă și 8 Memorează	4 Încarcă și 4 Memorează
Registre, cuvinte pe procesor				
Scalar	64B, 64T, 8A, 8S	8A, 8S	64B, 64T, 8A, 8S	256
Vector	512=8×64	512=8×64	512=8×64	—
Alte caracteristici		Fără înlănțuire; memorie locală (128 Kbyte/proc)		

IBM 3090S	Fujitsu/Amdahl	VP400E Hitachi S-830-80	NEC SX 2
—	Aprilie 1982	August 1982	1983
Mai 1986	Noiembrie 1983	Noiembrie 1983	Mai 1985
1, 2, 3, 4, 5, 6	1	1	1
133, 3 Mflops	1710 Mflops	3000 Mflops	1300 Mflops
4600 Mflops	1710 Mflops	3000 Mflops	1300 Mflops
15 ns	7 ns (vector) 14 ns (scalar)	4 ns (vector) 8 ns (scalar)	6 ns
0,5 Gbyte	0,25 Gbyte	0,25 Gbyte	1 Gbyte
2 Gbyte	1 Gbyte	12 Gbyte	8 Gbyte
ECL	ECL	ECL	ECL
612/2360 porti	400/1500 porti	2500/5000 porti	1000 porti
—	350 ps	200 ps/250 ps	250 ps
NMOS 1 M DRAM	256 K SRAM	256 K SRAM bi-CMOS 20 ns 1 M MOS DRAM	256 K MOS SRAM; 1 M MOS DRAM
80 ns	35 ns	120 ns	—
16—128; —	32; —	64; —	6; 1
4,5; —	3; —	4,5 ; —	15,9; 1300
500; —	96; —	288; —	95,4; 1300
Apa (100—132 pastile/modul)	Aer	Aer	Apa (logica) Aer uscat (memorie)
MVS, VM, AIX/370	MSP/MVS/XA/UTS/M	—	SXOS
Fortran 77, APL-2 APL—2	Fortran 77	Fortran 77	Fortran 7
6 Adun/Scd/Logic 6 Înmulţire/Imp.	4 Înmulţire 4 Împărţire 4 Adunare/Logic 4 Mascare	4 Înmulţire 1 Împărţire 4 Adunare/Logic 4 Înmulţire VM 4 Adunare VM	4 Înmulţire/Împărţire 4 Adunare 4 Logic 4 Deplasare
6 Încărcare sau 6 Memorare (combinaţii)	2 Încărcare sau 1 Încărcare 1 Memorare	8 Încărcare sau 4 Încărcare, 5 Memorare (date contigue)	8 Încărcare şi 4 Memorare
16 GP, 4FP	—	—	128
2K = 8×256	16 K (configurabilă dinamic) 2048 elemente/vector	16K = 32×512	16 K
	Tampon (cache)		Tampon scalari.

Tabelul P3. **Caracteristicile supercalculatoarelor anunțate după 1988 și ale unor supercalculatoare în faza de proiect.**

	NEC—SX—X/SX—3/44	Fujitsu VP2000/VP 2600/20	SSI	Cray-3	Cray-4	Cray-C-90
Anunțat	Aprilie 1989	Decembrie 1988				
Prima instalare	1989	1990	199X	199X	199X	199X
Număr procesoare	1, 2, 4	1,2	16, 32, 64	16	64	16
Performanțe max.						
Pe procesor	5,5 Gflops	2 Glops	—	1 Gflops	2 Gflops	1 Gflops
Total vîrf	2 Gflops	4 Gflops	—	16 Gflops	128 Gflops	16 Gflops
Durata ciclului	2,9 ns	4 ns	2 ns	2 ns	1 ns	4 ns
Cap. max. memorie	2/16 Gbyte	2/8 Gbyte				
Extensie memorie						
Tehnologie	ECL	ECL		GaAs, ECL,	GaAS	
Logica	20000 porți/pastilă	15000 porți/pastilă		500 porți/pastilă		
Întîrzierea pe poartă	70 ps	80 ps				
Memoria principală	bi-CMOS 256 K SRAM (20 ns)	SRAM 1 M (35 ns)		MOS		
Extensie mem.	DRAM 1M (150 ns)					
I/E						
Număr de căi	256+48+8	16—128				
Debit max./cale Mbytes/s	3, 4, 5, 6, 9/20/100	9 (fibră optică)				
Debit max. agregat Mbytes/s	1068	1000				
Răcirea	Apă (logica) Aer (memoria)	Apă				
Sistem de operare	Super-UX, SXOS	MSP, UTS/M	Unix	Unicos	Unicos	Unicos
Limbaje compilatoare	Fortran 77/SX C	Fortran	Fortran, C++?	C Fortran, C++?	C Fortran, C C++?	Fortran, C C++?
Caracteristici						
Benzi de asamblare (vectori)	8 Înmulțire /Deplasare 8 Adunare/Boolean	Dual scalar/ Mono vector pe procesor				
Registre						
Scalar	512 cuvinte					
Vector	72 K cuvinte	16 K cuvinte			23	
Alte caracteristici	Memorie tampon (cache).					

● *Tipuri reprezentative de supercalculatoare de capacitate medie-minisupercalculatoare pentru multiprelucrare, prelucrare paralelă, prelucrare de vectori.*

Folosind notaţiile din paragraful anterior, caracteristicile unor asemenea tipuri de calculatoare sînt prezentate în Tabelul P4.

Tabelul P4, **Tipuri de superacalculatoare de capacitate medie**

Modelul	Arhitectură/ Configuraţie	Nr. max. de procesoare	Tip procesor tehnologie	Cap. max.a memoriei	Performanţe
Alliant FN8	MP cu SM conexiune pe magistrală	8 proc. 12IP	M 68010 Gate array	4 Mw	94 Mflops, 35,6 Mips
ELXSI 6400	MP cu SM conexiune pe magistrală	12 proc.	ECL/VLSI	192 MB	156 Mips
UNCORP multimex	MC cu SM şi conexiune pe magistrala	20 proc.	NS 32032	16 Mw	35 Mips
Flexible 32	MC cu DM şi magistrale VME	20 proc. cabinet	NS 32032 M 68020	20 Mw	35 Mips/ procesor.
Convex CM/XP	UP cu hardware pentru vectori	4 proc. 5 IOP	CMOS/VLSI Gate array	16 Mw	20 Mflops; 25,5 Mips
SS C-40	UP cu BA multiple	1 proc.	ECL/LSI	4 Mw	4,4 Mflops; 18 Mips
IPS C-VX	MC cu DM şi hipercub	128 noduri de calcul	80286/80287	32 Mw	15 Mflops
Ametek S-14	MC cu DM şi conexiuni hipercub	256 proc.	80286/80287	32 Mw	15 Mflops
NCUBE/10 (fig. P5)	MC cu DM şi conexiuni hipercub	1024 proc.	custom VLSI	S 72 MB	500 Mflops; 24 IPS
FPS-T (fig. P6)	MC cu DM şi conexiuni hipercub	4096 proc.	Tranputer CMOS	1 MB/proc.	16 Mflops; 7,5 Mips/nod.

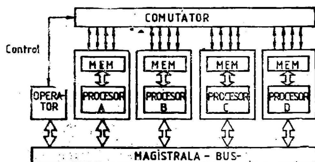


Fig. P5 Supercalculator de tip „desk-top", bazat pe procesoare de tip transputer INMOS-T8,

În continuare se prezintă produsele recent anunțate ale unor firme.

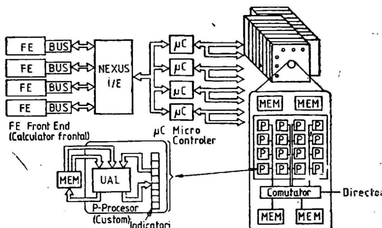


Fig. P6. Arhitectura hipercub pentru mașina de tip „connectionist".

● *HNSX Supercomputers, Inc.*, (*Burlington, MA*) s-a adresat pieței de supercalculatoare extrem de rapide și fiabile prin introducerea Seriei *SX-X*, care folosește o arhitectură bazată pe banda de asamblare pentru manipularea rapidă a vectorilor și scalarilor. Seria *SX-X* operează sub o implementare a sistemului de operare UNIX, care poartă numele de SUPER-UNIX. Aceasta permite integrarea seriei cu alte sisteme bazate pe UNIX, cît și utilizarea unei vaste biblioteci de programe aplicative pentru analiza structurală, analiza cîmpurilor magnetice, calcule numerice și simulare, studiul curgerii fluidelor, testarea mediului, prelucrări de imagini, grafică etc. Modelul de la capătul superior SX-X4. poate realiza o viteză de 22 Glops.

● *MasPar Computer* (*Sunnyvale, CA*) a anunțat recent un supercalculator cu paralelism masiv destinat calculelor științifice și inginerești. Familia *MP-1* include 8 configurații cu prețuri care încep de la 170 000 dolari, pentru un sistem cu 1024 de procesoare lucrînd în paralel, pînă la 810 000 dolari, pentru configurația MP1216, care dispune de 16384 de procesoare.

● *Evans & Sutherland Computer Division* (*Mountain Kiew, CA*) a anunțat în 1989 sistemul *ES-1*. Arhitectura lui este bazată pe configurații cu 2–8 procesoare, fiecare procesor dispunînd de 16 unități de calcul. Fiecare unitate de calcul operează independent ca un calculator paralel puternic. Sistemul este destinat rezolvării problemelor cu caracter științific și ingineresc în domenii ca: modelarea moleculelor, inginerie aerospațială și transporturi. Costurile, în funcție de configurație, variază între 2 și 8 milioane de dolari.

● *Myrias Computer Corporation* (*Boston, MA*) a lansat recent sistemul paralel *Myrias SPS-2*, pentru aplicații industriale, științifice și inginerești în domeniile dinamicii fluidelor, modelării moleculelor, pre-

lucrării de imagini etc. Avînd o structură modulară, se poate extinde uşor de la maximum 64 procesoare, la peste 1000 procesoare. Software-ul de sistem asigură gestiunea automată a memoriei şi resurselor. Configuraţia SPS-2, cu 64 de procesoare, costă circa 0,5 milioane de dolari, în timp ce configuraţia cu peste 1000 de procesoare costă circa 10 milioane de dolari.

● *Apollo Computer Inc.*, *(Chelmsford, MA)* realizează *Series 10000 Visualization System (Series 10000 VS)*, care combină viteza supercalculatoarelor cu grafica 3-D şi arhitectura RISC. Ca aplicaţii se au în vedere dinamica fluidelor, analiza elementului finit, modelarea moleculelor, prelucrarea imaginilor în medicină, analiza datelor seismice, proiectarea în electronică, animaţia imaginilor. In *Series 10000 VS*, prelucrările au loc simultan cu vizualizarea rezultatelor, folosindu-se procesoare de imagini 3-D RISC. Ele încorporează execuţia paralelă, cu trasarea de pixeli într-un singur ciclu, fără microprogramare. Sistemul poate afişa peste 1 milion de vectori 3-D (virgulă mobilă)/s, şi circa 100000 poligoane 3-D/s.

● *Stardent Computer Inc.*, *(Sunnyvale, CA)* a lansat în 1989 sistemul grafic-supercalculator *Stardent 3000*, bazat pe tehnologia RISC. În configuraţia completă cu 4 procesoare *Mips Computer R300/3010*, operînd la 32 MHz, poate asigura o viteză de 128 MIPS şi 192 Mflops. *Stardent 3000* are o structură modulară şi este destinat aplicaţiilor inginereşti care necesită o mare viteză de calcul, cît şi capabilităţi grafice interactive 3-D. Costul configuraţiei minime este de 89000 de dolari.

● *NCUBE (Beaverton, OR)* oferă supercalculatorul *NCUBE 2 Scalar*, bazat pe un paralelism masiv. Sistemul foloseşte microprocesoare VLSI, care integrează pe o pastilă un calculator pe 64 de biţi. În funcţie de specificul aplicaţiei poate fi configurat ca număr de procesoare. Într-o configuraţie cu 8192 procesoare, *NCUBE 2* realizează 60 Gips. Capacitatea maximă a memoriei este de 512 G bytes. Aceasta permite abordarea problemelor de prognoză a vremii şi, în general, a problemelor în care se manipulează cantităţi mari de date. Alte domenii de aplicaţii : reţele neuronale, robotică, prelucrarea semnalelor, dinamica fluidelor, prelucrarea tranzacţiilor, analiza structurală şi chimică etc. Costul unui sistem minim este de circa 395000 de dolari.

● *CONVEX Computer Corporation*, (*Richardson*, *TX*) a proiectat un sistem de tip supercalculator accesibil ca preţ oamenilor de ştiinţă şi inginerilor din organizaţii ce dispun de resurse financiare modeste. Apelînd la numeroase standarde industriale, sistemul permite o integrare uşoară în mediul utilizatorului. Pentru a întări sistemul *C. Series* de supercalculatoare integrate, pentru prelucrări de scalari, vectori, cît şi pentru prelucrări paralele, s-a introdus procesorul *Enhanced Scalar Processor (ESP)*, care este compatibil la nivel obiect cu seria *C* Canalul Integrat pentru Disc *(Integrated Disk Channel — IDC)* asigură funcţia de *server* pentru fişiere şi baze de date, la un nivel înalt de performanţe. Costuri : *ESP* — 75000 dolari, IDC — 48 dolari.

● *Aptec Computer System, (Portland, OR)* a proiectat sistemul *Aptec IOC Computer*, folosit pentru conexiuni de I/E. Se au în vedere sis-

temele frontale pentru telemetrie, procesoarele specializate şi procesoarele de imagini. Sistemul poate prelucra şi transfera cantităţi mari de date în timp real, pentru aplicaţii asistate de calculator.

● *Evaluarea creşterii de viteză în rezolvarea problemelor, folosind supercalculatoarele în locul calculatoarelor seriale.*

Aspectele creşterii vitezei de prelucrare, prin folosirea calculatoarelor paralele au fost extrem de controversate. Astfel, în anul 1967, Gene Amdahl propune o formulă prin care se stabileşte o limită a creşterii de viteză, n, în execuţia unui program de către P procesoare, în raport cu execuţia aceluiaşi program de către un singur procesor.

Fie N lungimea totală a programului, din care S lungimea de natură pur secvenţială. Creşterea de viteză n va fi dată de formula:

$$n = \frac{\textit{timpul de calcul pentru un procesor}}{\textit{timpul de calcul pentru } P \textit{ procesoare}} = \frac{N}{S+(N-S)/P},$$

Trecînd la limită se obţine:

$$\lim_{P \to \infty} n = \frac{N}{S}$$

Rezultă că, indiferent de numărul procesoarelor utilizate, creşterea de viteză este limitată de caracteristicile intrinseci ale programului.

În abordarea performanţelor prelucrării paralele se pot evidenţia două căi distincte.

Prima cale presupune soluţionarea cu viteză mai mare a unei probleme de dimensiuni fixe. Mărirea numărului de procesoare va conduce la reducerea sarcinii atribuite unui procesor şi la creşterea timpului necesar comunicaţiilor între procesoare. Astfel, va rezulta *figura de merit „creşterea de viteză la dimensiune fixă"*, care este guvernată de formula lui Amdahl.

A doua cale consideră soluţionarea unei probleme de dimensiune maximă, într-un interval dat de timp, prin atribuirea fiecărui procesor a unei sarcini fixe; sarcina totală se va scala cu numărul procesoarelor. Ca rezultat se va obţine *figura de merit „creşterea scalată de viteză"*. În acest caz se presupune că fiecare procesor comunică numai cu vecinii săi cei mai apropiaţi, raportul timp de comunicaţii/timp de calcul fiind constant. Creşterea scalată de viteză nu mai este guvernată de formula lui Amdahl, rezultatul fiind caracteristic multor aplicaţii practice. În sprijinul celor arătate mai sus se va da exemplul soluţionării unei ecuaţii cu derivate parţiale într-un domeniu acoperit cu o grilă cu $N \times N$ noduri, folosind un algoritm care actualizează nodurile grilei de la vecinii cei mai apropiaţi.

În primul caz se consideră creşterea de viteză la dimensiune fixă. Timpul total de calcul este $a.N^2$, unde a reprezintă intervalul de timp pentru calculul într-un nod. Timpul necesar schimbului de date între două noduri vecine ale grilei este egal cu b; comunicaţiile se realizează în paralel.

Dacă problema se poate împărţi în mod egal între P procesoare, rezultă o creştere de viteză :

$$n = \frac{a \cdot N^2}{b + a \cdot N^2/2}$$

Trecînd la limită se obţine :

$$\lim_{P \to \infty} n = \frac{a \cdot N^2}{b}.$$

Rezultatul este independent de P.

În al doilea caz dimensiunea problemei se scalează cu numărul de procesoare : $N^2 = c \cdot P$, unde c este numărul de noduri atribuite unui procesor. Inlocuind în expresia lui n, de la primul caz, rezultă o creştere scalată de viteză, s, care este liniară cu P :

$$s = \frac{P}{1 + b/a \cdot c}$$

Abordări recente ale grupului Sandia, care a folosit un calculator *NCUBE/10* (hipercub), cu 1024 procesoare, au condus la o creştere scalată de viteză de 400 de ori în raport cu sistemele uniprocesor.

● *Aspecte privind algoritmii paraleli, limbajele de programare paralelă şi tehnicile de compilare a programelor paralele.* După cum se ştie, problemele de bază ale calculului paralel se referă la : elaborarea algoritmilor cu paralelism inerent (pornind de la clase de aplicaţii), proiectarea limbajelor de programare paralelă, stabilirea unor arhitecturi hardware şi a unor sisteme de operare adecvate.

Eficienţa sistemelor paralele este condiţionată de o serie de factori :

— identificarea unei probleme care posedă un paralelism intrinsec ;

— elaborarea unui algoritm corespunzător ;

— aplicarea/maparea algoritmului într-o structură hardware/software convenabilă.

Pentru a ilustra diversele tehnici de programare paralelă se consideră problema calculului integralei definite a unei funcţii de o singură variabilă. Spaţiul cuprins între reprezentarea grafică a funcţiei şi axa absciselor este acoperit cu un număr de dreptunghiuri. Suma suprafeţelor dreptunghiurilor, în funcţie de pasul de discretizare pe axa absciselor, aproximează mai precis sau mai puţin precis, valoarea integralei definite.

Se vor considera mai multe tehnici : calculul secvenţial folosind un sistem uniprocesor ; calcul paralel pe sisteme multiprocesor cu memorie partajată ; calcul paralel pe sisteme multiprocesor cu memorie distribuită (transfer de mesaje) şi calculul paralel cu sisteme multiprocesor folosind anticipările (futures).

● *Programul pentru sistemul uniprocesor* descompune suprafaţa în dreptunghiuri foarte mici, calculează aria fiecărui dreptunghi şi în final adună ariile calculate.

● *În varianta cu memorie partajată*, se generează o serie de taskuri pentru a calcula aria fiecărui dreptunghi şi a o aduna automat la suma curentă. Suma curentă este stocată ca o variabilă în memoria partajată; cînd toate taskurile sînt terminate valoarea ei este returnată. Taskurile care calculează aria fiecărui dreptunghi sînt asignate nodurilor procesoare pe măsură ce termină taskurile anterioare. Atribuirile sînt realizate de către un generator de taskuri, care are aceeaşi funcţie ca şi bucla în versiunea serială ce calculează şi sumează în secvenţă ariile dreptunghiurilor. Generatorul de taskuri maximizează utilizarea procesorului, prin alocarea dinamică a taskurilor procesoarelor disponibile: ele reprezintă expresia directă şi succintă a paralelismului inerent în program.

● *În versiunea cu transferuri de mesaje*, un procesor „părinte" trimite mesaje către toate celelalte procesoare din sistem, (procesoare „copii"), dîndu-le date pentru fiecare dreptunghi a cărui arie trebuie calculată. Cînd un procesor-copil a calculat aria unui dreptunghi, el trimite un mesaj, ce conţine rezultatul, înapoi către părinte, care adună acest rezultat la suma curentă/parţială şi apoi trimite procesorului-copil datele pentru calculul ariei unui nou dreptunghi. Versiunea cu transferuri de mesaje este mai complexă decît versiunile anterioare, deoarece programatorul trebuie să gestioneze într-o manieră explicită paralelismul. Programatorul trebuie să aibă în vedere doi algoritmi: algoritmul care rulează pe procesorul-părinte şi algoritmul care se execută pe procesorul-copil.

● *Versiunea bazată pe anticipări (futures)*, care reprezintă o nouă tehnică pentru programarea paralelă, porneşte calculele şi primeşte locurile de păstrare a rezultatelor,; în aceste locuri se plasează valorile reale, pe măsură ce acestea se calculează. Anticipările sînt alocate pentru fiecare dreptunghi aflat sub curbă, iar procesoarele din sistem încep calculul valorilor adevărate pentru aceste anticipări. Odată ce o anticipare a fost calculată, ea este adunată la suma curentă. Cînd toate anticipările sînt alocate simultan, programul poate întîrzia evaluarea rezultatului pînă la lansarea ultimului calcul. Astfel, anticipările care sînt lansate mai întîi vor avea mai mult timp pentru calcul, înainte ca programul să facă suma rezultatelor returnate. Deoarece programul va aştepta mai puţin timp pentru ca valorile reale să fie plasate în locurile de păstrare, el va fi mult mai eficient.

● *Implementarea algoritmilor paraleli, rapizi şi eficienţi, pe supercalculatoare necesită limbaje de nivel înalt, flexibile în specificarea diferitelor forme de paralelism şi eficiente în utilizarea lor pe diverse tipuri de calculatoare paralele. În acest sens se evidenţiază trei soluţii.*

Prima soluţie constă n folosirea compilatoarelor vectorizante, care generează cod paralel, plecînd de la programele scrise pentru prelucrarea serială. Astfel, sînt cunoscute compilatoarele vectorizante: CFF, pentru sistemele de tip Cray, şi KAP-205, pentru sistemul Cyber 205. Eficienţa acestei abordări este modestă.

A doua soluție pleacă de la extensia limbajelor curente Fortran, C etc., cu facilități capabile să suporte construcții multitasking și concurente. Astfel, diferiți algoritmi paraleli au fost descriși în Fortran extins, pentru construcții multitasking (Cray X-MP-multiprocesor) și în C extins, pentru construcții concurente (IPSC-multicalculator). Eficiența acestei abordări este moderată.

A treia soluție se bazează pe elaborarea unor noi limbaje capabile să permită programarea prelucrării paralele. Aceste noi limbaje, care suportă prelucrarea paralelă au la baza limbajul ALGOL (Ada, Linda, Pascal concurent, Modula 2, Occam), limbajul LISP paralel/limbaje logice (Multilisp, Prolog concurent), limbaje funcționale paralele (Parafal).

Pentru a specifica explicit tipurile de proceduri utilizate în diferitele moduri ale calculului paralel, în unele din noile limbaje se folosește o nouă construcție numită moleculă.

Algoritmii paraleli și supercalculatoarele suportă diverse moduri de calcul, în timp ce limbajele de programare suportă unul sau cel mult două asemenea moduri.

Limbajele bazate pe ALGOL explorează un spectru larg, de la Ada inițial proiectat pentru sistemele care operează secvențial (limbajul avînd însă și facilități de prelucrare paralelă), pînă la Linda, special proiectat pentru prelucrarea paralelă (în care un program paralel se comportă spațial și temporal ca un „ansamblu neordonat de procese și nu ca un graf de procese).

Sistemul Linda constă din operatori care pot orienta oricare limbaj gazdă (Fortran sau C) într-un limbaj de programare paralelă. Ca limbaj autonom, el constă dintr-un nucleu pentru sincronizare în timpul execuției și dintr-un compilator. Paralelismul este asigurat, atît sub forma unei partiții de procese simultane, cît și prin multiplicarea unui singur proces.

LISP paralel este orientat pe prelucrări simbolice paralele, constituind primul candidat pentru aplicațiile de inteligență artificială, cu accent pe operații recursive, pe operații cu arbori și liste.

Programarea paralelă funcțională are la bază metodologii care permit maparea programelor pe topologii de prelucrare paralelă. Sistemul multiprocesor este tratat ca un singur calculator autonom, în care este mapat un program și nu ca un grup de procesoare independente, care efectuează comunicații complexe ce necesită, de asemenea, sincronizări complexe.

Limbajele funcționale garantează faptul că un program va conduce la același rezultat, indiferent de ordinea în care a fost executat. În limbajele funcționale paralelismul este implicit, fiind suportat de semantica dată. Limbajele funcționale reprezintă primii candidați pentru programarea mașinilor paralele.

● *Tendințe privind viitoarele supercalculatoare.*

Supercalculatoarele viitoare vor beneficia de noi tehnologii: GaAs, optice, neuroni artificiali, materiale supraconductoare, integrare pe scară ultralargă în 3D, împachetări și interconexiuni extrem de dense. În prezent se înregistrează o serie de realizări: folosirea GaAs (Cray-3 și unele calculatoare RISC); interconexiuni optice, de tip „crossbar" și rețele de'

porţi electronooptice; reţele neuronale pentru modelul „conecţionist" al calculului paralel (Mark III, bazat pe 8 procesoare M68010, care implementează 8100 elemente virtuale de procesare, cu 417000 conexiuni între ele). Practic există posibilitatea de realizare a unor reţele cu peste 10^6 elemente de procesare şi peste 100 milioane interconexiuni. În Tabelul P5

Tabelul P5, **caracteristici comparative ale calculatoarelor electronice, optice şi neuronale**

Caracteristici	Calc. Electronice	Calc. Optice ;	Calc. Neuronale
Timp de comutaţie	10^{-9}—10^{-11} s, în Si, GaAs, JJ	10^{-12}—10^{-15} s, în reţele de porţi optice.	10^{-10} s, electronice. 10^{-13} s optice 10^{-3} s, biologice.
Granularitate ; paralelismul procesorului	Mare ; 1—100 procesoare/sistem	Fină : paralelism masiv	Fină ; paralelism masiv
Banda de comunicaţii	10—1000 Mbits/s	1—100 Gbit/s	Foarte mare ; comunicaţii paralele.
Integrare fizică	10^4—10^5 tranzistori/ /pastilă.	10^2—10^6 porţi pe o reţea optică	10^2 neuroni electronici pe o pastilă de Si
Complexitatea comenzii	Comandă sincronă cu ceas numeric	Comanda sincronă/ asincronă fără întirzieri ale ceasului	Comanda distribuită, cu autoorganizare
Puterea consumată	10^2 W pe plachetă, la 10 MHz	10^{-1} W pe reţea, la 10 MHz.	Este funcţie de tehnologia de implementare ; electronică, optică.
Fiabilitate ; întreţinere	Dependente de arhitectură şi tehnologie.	Interferenţe reduse ; costisitoare.	Robustă datorită cooperării

se prezintă o serie de caracteristici ale calculatoarelor electronice, optice şi neuronale. De menţionat faptul că, prin granularitate se înţelege raportul între intervalul de timp necesar comunicaţiei şi intervalul de timp necesar calculelor paralele în cadrul execuţiei unui program paralel pe un sistem multiprocesor. În cazul unei granularităţi grosiere, prelucrarea paralelă se efectuează pe segmente mari de program, independente, care comunică rar între ele (sute de comunicaţii/s între procesoarele individuale). Aplicaţiile cu granularitate fină consumă mai mult timp pentru comunicaţii şi sincronizări (milioane de comunicaţii/s).

În ţările puternic industrializate se acordă o mare atenţie cercetărilor în vederea realizării de supercalculatoare. În Japonia şi în ţările din Europa de Vest au fost lansate programe de cercetare speciale, la nivel naţional. După cum se vede şi din Tabelul P6, în Japonia, în cadrul programului „High-Speed Computing", în perioada 1982—1990, s-au cheltuit circa

140 milioane de dolari, în vederea atingerii unor performanțe de circa 10 Gflops. În cadrul programului ESPRIT, în Europa de Vest, au fost lansate programele : Supernode I ; II, în perioadele 1984—1988 și 1988—1992 (25 milioane de dolari și respectiv — 30 milioane de dolari, pentru

Tabelul P6 **Programe naționale sau interfațări privind dezvoltarea unor noi supercalculatoare**

	Japonia	Europa/ESPRIT	
Nume	High-Speed Computing.	Supernode I ; II	Suprenum-ISIS
Perioada	Ian 1982—Mar. 1990	1984—1988 1988—1992	1988—1992
Efort financiar	$140 mil.	$ 25 ; 30 mil.	$ 1,5 mil. faza 0 $ 40 mil. total
Performanțe	10 Gflops	40—1500 Mflops 0,2—10 Gflops	40—120 Gflops.
Obiective curente	Noi dispozitive (JJ, HEMT) Sisteme dedicate pentru prelucrări de imagini : Arhitecturi paralele ; Algoritmi paraleli ; Limbaje paralele.	Prelucrare paralelă.	
Participanți	Electrotechnical Laboratory Hitachi Fujitsu NEC	Thom EMI Telmat INMOS RSRE ARSIS Univ. of Grenoble Univ of Southampton	Germania. Franța
Metode pentru introducerea în practică.	Încorporate în eforturile participanților pentru realizarea de supercalculatoare ; arhitecturi specializate pentru necesități naționale.	Crearea unei noi corporații industriale	

a atinge performanțele de 1,5—10 Gflops) și Suprenum-ISIS, în perioada 1988—1992 (circa 41,5 milioane de dolari, în vederea realizării unor supercalculatoare cu viteză de 40—120 Gflops).

* * *

Traducerea lucrării Calculatoare Paralele, ediția a II-a, autori profesorii Hockney R. W. și Jesshope C. R., constituie un eveniment editorial deosebit, care umple un mare gol în literatura tehnică de specialitate, în domeniul calculatoarelor, în țara noastră.

Prima ediție a cărții Calculatoare Paralele, apărută în anul 1981, a fost folosită parțial, de către autorul acestor rînduri, ca material de referință, în predarea cursului „Structura Sistemelor Numerice de Prelucrare a Datelor", La Facultatea de Automatică și Calculatoare, din Institutul Politehnic București, pentru studenții direcției de aprofundare „Construc-

ţia calculatoarelor". Ca suport de curs au mai fost folosite, printre altele, şi lucrările :

1. Stone H., (editor) *Introduction to Computer Architecture*. First edition-1975. Second edition — 1980. Science Research Associates.
2. Hwang K., Briggs F. *Computer Architecture and Parallel Processing*. Mc Graw Hill Book Company, 1984.
3. Stone H. *High — Performance Computer Architecture*. Addison — Wesley Publishing Company, 1987.
4. Fernbach S. *Supercomputers Class VI Systems. Hardware and Software*. North — Holland. 1986
5. * * * *Special Issue on Supercomputing IEEE Transactions on Computers*. C. 36. No. 12. December 1987
6. * * * *Special Issue on Multicomputers. Computer*. August 1988.
7. * * * *Proceedings of International Conferences on Parallel Processing*. Computer Society Press.
8. Siewioreck D. (editor) *Computer Structures : Principles and Examples* Mc Graw Hill Book Company. 1982.

De menţionat faptul că lucrarea [2] reprezintă una dintre cele mai reuşite lucrări în domeniu, oferind multe detalii de proiectare.

De la publicarea lucrării Parallel Computers, ediţia a II-a, în 1988, domeniul supercalculatoarelor a continuat să se dezvolte într-un ritm susţinut. Unele aspecte ale elementelor apărute după elaborarea lucrării au fost prezentate în POSTFAŢĂ. De asemenea, s-a căutat completarea referinţelor bibliografice, pe cît a fost posibil.

Bibliografie. Supercalculatoare.

1. Andre F., Verjus J. P. (editori). *Hypercube and distributed computers*. (Proccedings of the First European Workshoop, Rennes, France 4—6 Oct. 1989). North-Holland. 1989. ISBN 0—444—88086—0

Conţinut : *Algoritmi numerici : Ecuaţii cu derivate parţiale ; Unelte şi Medii* (de dezvoltare) ; Algoritmi Fundamentali în Algebra Liniară ; Limbaje şi sisteme de Operare ; Reţele Neuronale ; Algebra Liniară şi Calcul Ştiinţific ; Reţele de Interconectare.

Seria : **,,Special Topics in Supercomputing"**. North—Holland.

Editori : Rodrigue G., Fernbach S., Michael G.

2. Dongarra J. J. (editor). *Experimental Parallel Computing Architectures*. Volume 1. 1987 North—Holland. ISBN 0—444—70234—2.
3. Schonhauer W. (editor). *Scientific Computing on Vector Computers*. Volume 2. 1987. North—Holland. ISBN 0—444—70288—1.
4. te Riele H. J. J. van der Vorst H. A. (editori). *Algorithms and Applications on Vector and Parallel Computers*. Volume 3. 1987. North — Holland. ISBN 0—444—70322—5.
5. Martin J. L. (editor). *Performance Evaluation of Supercomputers*. Volume 4. 1988. North—Holland. ISBN 0—444—70448—5.

conţinut : Performanţe ; Măsurători şi Metrice ; Metode, Modele şi Direcţii (de Cercetare)

Seria : **,,Advances in Parallel Computing"**. North—Holland

Editori : Joubert G. R., Peters F. J., Feilmeier M., Schendel U.

6. van der Vorst H. A., van Dooren P. (editori). *Parallel Algorithms for Numerical Linear Algebra*. North — Holland. 1989.

Conţinut : Algoritmi pentru masive sistolice ; Sisteme cu transfer de mesaje ; Algoritmi pentru sisteme paralele cu memorie partajată ; Proiectarea algoritmilor rapizi şi implementări pentru supercalculatoare vectoriale.

7. Joubert G. R., Evans D. J., Peters F. T. (editori). *Parallel Computing '89*. Volume 2. Nort — Holland. 1989.

8. Consard M., Robert Y., Quinton P., Raynal M. (editori). *Parallel & Distributed Algorithms*. (Proceedings of the International Workshop., Chateau de Bonas, Gers, France, 3—6 october 1988). North — Holland. 1989. ISBN 0—444—87367—8.

Conținut : Algoritmi Paraleli ; Sortare și Căutare Paralelă ; Algoritmi Distribuiți ; Metode de paralelizare.

9. Delhye J. L., Gelenbe E. (editori). *High Performance Computing*. (Proceesings of the International Symposium on High Performance Computing, Montpellier, France, 22—24 March 1989). ISBN 0—444—88034—7.

Conținut : Unelte și Algoritmi Numerici ; Arhitecturi Paralele ; Aplicații în DinamicaFluidelor ; Aplicații în Chimie ; Aplicații în Fizică ; Noi omenii de Aplicații ; Educația și Pregătirea (specialiștilor).

10. Hasegawa T. Takagi H. Takahashi Y. (editori) *Performance of Distributed and Parallel Systems*. (Proceedings of the IFIP TC7 WG7. International Seminar, Kyoto, japan, 7—9 December 1988). Notrh—Holland. 1989. ISBN 0—444—87497—6.

Conținut : Sisteme Distribuite, ISDN, Multiprocesoare, Sisteme cu integrare. Sisteme cu cozi de așteptare. Rețele de cozi de așteptare. Rețele Perti, Protocoale pentru Acces Multiplu. Sisteme Software, Prelucrare Paralelă. Rețele de Interconectare.

11. Winter S. Schumany H. (editori). *Supercomputers :Technology and Applications* (Proceedings of the Fourteenth EUROMICRO Symposium on Microprocessding and Microprogramming (EUROMICRO' 88) Zurich, Switzerland, 29 August — 1 September 1988).

Conținut. Microelectronică, Inginerie Software. Prelucrare Paralelă, Aplicații. Muzica pe calculator.

12. Wright M. (editor). *Aspects of Computation on AsynchronousParallel Processors*. Proceedings of the IFIP WG2. 5 Wotrking Conference, Stanford, CA, USA, 22—26 August 1988). North — Holland. 1989. ISBN 0—444—87310—4.

Conținut : Aplicații Științifice Eficiente, Limbaje și Programe, Biblioteci Unelte și Medii (de dezvoltare), Limbaje de proiectare, Sesiuni Deschise, Proiectare de Sistem.

Reviste — Periodice

13. Vichenevetschi R. Stepleman R. (editori). *Applied Numerical Mathematics*. North—Holland ISSN 0168—9274.
14. Hertzberger L. O. Ward S. (editori). *Future Generation systems*. North—Holland. ISSN 0167—739×
15. Feilmeier M., Joubert G. R. Schendel U., Hiromoto R., Evans D. J., Hoshino T. editori. *Parallel Computing*. North — Holland. ISSN 0167—8191.
16. Reiser M., Bux W. editori. *Performance Evaluation*. North—Holland. ISSN 0166—5361.
17. * * * *Computer Phisics Communications*. North—Holland. ISSN 0010—4655.
18. * * * *Computer Physics Reports*. North—Holland. ISSN 0167—7977.
19. * * * Integration, The VLSI Journal. North—Holland. ISSN 0167—9260.

20. Petrescu A. *Structuri paralele sistolice larg integrate pentru soluționarea ecuațiilor cu derivate parțiale*. Lucrările Conferinței SAII. 1987. București.
21. Petrescu A., *Considerații preliminare privind proiectul unui sistem paralel de calcul*. Lucrările Conferinței INFOTECH. 1988. București.
22. Petrescu A., (și colectiv) *Conectarea magistrală-magistrală a microsistemului Felix PC cu microsisteme MIND*. Simpozionul de Microprocesoare, Microcalculatoare și Aplicații. 1989. IPB. București.
23. Petrescu A. *Sisteme sistolice de prelucrare a datelor*. Lucrările impozionului ,,Calculatoarele Electronice din Generația a Cincea". Editura Academiei Romane. 1985. București.

Anexă

Sintaxa ASN — O notație structurală în stil algebric. Se definește în continuare, pe baza formei Backus Naur (BNF) așa cum este ea folosită în raportul ALGOL60 Backus et al 1960), sintaxa notației structurale utilizate pentru arhitectura calculatoarelor. Parantezele unghiulare (⟨⟩) delimitează termeni metalingvistici, iar barele verticale (|) separă alternativele (se citește ca „sau"). Două puncte de două ori urmate de egal (: : =) trebuie citit ca „poate fi".

A.1 Diverse

⟨gol⟩ : : =
⟨cifră⟩ : : = 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨literă mică⟩ : : = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
⟨pipeline⟩ : : = p | ⟨gol⟩
⟨SI prefix⟩ : : = K | M | G | T⟨gol⟩
⟨întreg fără semn⟩ : : = ⟨cifră⟩ : ⟨ întreg fără semn⟩ ⟨cifră⟩
⟨putere⟩ : : = ⟨întreg fără semn⟩ : ⟨gol⟩ | ⟨literă mică⟩
⟨multiplicator⟩ : : = ⟨literă mică⟩ | ⟨întreg fără semn⟩ ⟨SI prefix⟩
⟨întreg fără semn⟩ ⟨întreg fără semn⟩ | ⟨comentariu]
⟨factor⟩ : : = ⟨multiplicator⟩ ⟨putere⟩ | ⟨factor⟩ * ⟨factor⟩
⟨comentariu⟩ : : = (⟨orice secvență de simboluri⟩) | ⟨gol⟩
⟨separator⟩ : : = ;
⟨aserțiune⟩ : : = ⟨definiție⟩ | ⟨definiție de cale⟩ | ⟨structură⟩

Exemple

⟨gol⟩ : : =
⟨cifră⟩ : : = 3 ; 9
⟨literă mică⟩ : : = c ; z
(pipeline⟩ : : = ; p
⟨întreg fără semn⟩ : : = 34 ; 128
⟨putere⟩ : : = ; 2 ; s
⟨multiplicator⟩ : : = n ; 16 ; 8 G ; 9,5
⟨factor⟩ : : = n * m ; 64^2 ; n * 128 * 32^2
⟨comentariu⟩ : : = ; (ECL bipolar)

A.2 Unități E

⟨simbol E⟩ : : = B : Ch | D | E | F | IO | P | U | S
⟨identificator E⟩ : : = ⟨simbol E⟩ ⟨pipeline⟩ | ⟨identificator E⟩ ⟨cifră⟩
⟨timp de lucru în ns⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨număr de biți al operandului⟩ : : = ⟨multiplicator⟩
: ⟨număr de biți al operandului⟩, ⟨multiplicator⟩ | ⟨comentariu⟩
⟨unitate E⟩ : : = ⟨identificator E⟩ ⟨timp de lucru în ns⟩ / ⟨număr de biți și op.⟩ ⟨comentariu⟩

Exemple

⟨identificator E⟩ : : = E ; F12 ; Bp46
⟨timp de lucru în ns⟩ : : = t ; (4 milisecunde)
⟨număr de biţi ai operandului⟩ : : = b ; (4 octeţi) ; 16, 32

⟨unitate E⟩ : : = E ; F12 $^{100}_{32}$ (*) ; L (omega)

A.3. Unităţi M

⟨simbol M⟩ : : = M | O
⟨identificator M⟩ : ⟨= simbol M⟩ ⟨pipeline⟩ | ⟨identificator M⟩
⟨cifră⟩
⟨timp de acces în ns⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨număr de cuvinte⟩ : : = ⟨multiplicator⟩ *
: ⟨număr de cuvinte⟩ ⟨număr de cuvinte⟩ | ⟨comentariu⟩ *
⟨biţi accesaţi din cuvînt⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨dimensiunea memoriei⟩ : : = ⟨număr de cuvinte⟩ ⟨biţi accesaţi din cuvînt⟩ | ⟨comentariu⟩

⟨unitate M⟩ : : = ⟨identificator M⟩ $^{\langle\text{timp de acces în ns}\rangle}_{\langle\text{dimensiunea memoriei}\rangle}$ ⟨comentariu⟩

Exemple

⟨identificator M⟩ : : = Mp ; M1 ; M2 ; M3 ; O16
⟨timp de acces în ns⟩ : : = 100 ; (4 milisecunde)
⟨număr de cuvinte⟩ : : = n* ; (1, 2 sau 4 Mocteţi) * ;
⟨biţi accesaţi din cuvînt⟩ : = b ; 32
⟨dimensiunea memoriei⟩ : : = n*b ; n *32 ;; n* 32 ; 2K*8*64

⟨unitate M⟩ : : = M ; O16 $^{100}_{n*32}$; M_{2K*8} (2716EPROM) ; $M_{8*64*64}$

A.4 Calculatoare

⟨simbol de control⟩ : : = I | Iv | C Cv
⟨identificator de control⟩ : : ⟨simbol de control⟩ ⟨pipeline⟩ | ⟨identificator de control⟩
⟨cifră⟩
⟨perioada ceasului în ns⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨biţii instrucţiunii⟩ : : = ⟨multiplicator⟩
:⟨biţii instrucţiunii⟩ ⟨multiplicator⟩ | ⟨comentariu⟩
⟨număr de fluxuri⟩ : : = ⟨multiplicator⟩ * : ⟨comentariu⟩ * | ⟨gol⟩
⟨fluxuri de instrucţiuni⟩ : : = ⟨număr de fluxuri⟩ ⟨biţii instrucţiunii⟩
⟨conectivitate⟩ : : = ⟨multiplicator⟩ — n n | ⟨comentariu⟩
⟨tipul controlului⟩ : : = a | h | l | r | ⟨comentariu⟩

⟨elemente comandate⟩ : : = [⟨structură⟩] $^{\langle\text{conectivitate}\rangle}_{\langle\text{tipul controlului}\rangle}$ ⟨gol⟩

⟨calculator⟩ : : = ⟨identificator de control⟩ $^{\langle\text{timp ceas în ns}\rangle}_{\langle\text{fluxuri de instr.}\rangle}$ ⟨comentariu⟩
⟨elemente comandate⟩

Exemple

⟨identificator de control⟩ : : = I ; Ivp3 ; Cv12
⟨elemente comandate⟩ : : = ;[$64\overline{P}$] ; [C1 [$64\overline{P}$], C2] ; [10Fp—4M]
⟨conectivitate⟩ : : = (structura 2D hexagonală) ; c—nn ; 2—nn
⟨tipul controlului⟩ : : a ; h ; l ; r ; ;

⟨calculator⟩ : : = C1 [$64^2\overline{P}$] $^{2-nn}_{1}$; C1[$16\overline{F}$ × $17\overline{M}$] ; $C4^{250}_{8}$;

Ivp (10Fp — 4M) ; Iv_{12} ; I_{50*64}(HEP)
$P_{16,32}$

A.5. Căi de date

⟨identificator de cale⟩ : : = H⟨intreg fără semn⟩ ⟨comentariu⟩ | ⟨comentariu⟩
⟨timp în ns pentru un cuvînt⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨biţi de date⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨biţi de adresă⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨număr de căi⟩ : : = ⟨multiplicator⟩ | ⟨comentariu⟩
⟨dimensiunea căii⟩ : : = ⟨număr de căi⟩ * {⟨biţi de date⟩ + ⟨biţi de adresă⟩} | ⟨număr de căi⟩ * ⟨biţi de date⟩ : ⟨biţi de date⟩ | ⟨biţi de date⟩ | ⟨biţi de adresă⟩

⟨magistrală de date⟩ : : = $\frac{\langle\text{timp de cuvînt în ms}\rangle}{\langle\text{dimensionea căii}\rangle}$ | ⟨ magistrală de date ⟩ —
| — ⟨ magistrală de date • ⟩

⟨conexiune în cruce⟩ : : = $\frac{\langle\text{timp pe cuvînt în ns}\rangle}{\langle\text{dimensiunea căii}\rangle}$ | ⟨conexiune în cruce⟩ ×
| × ⟨conexiune în cruce⟩

⟨conexiune⟩ : : = ⟨magistrală de date⟩ | ⟨magistrală de date⟩ ⟨identificator de cale⟩ ⟨magistrală de date⟩ | ⟨conexiune în cruce⟩ | ⟨conexiune în cruce⟩ ⟨identificator de cale⟩ ⟨conexiune în cruce⟩
⟨nici o conexiune⟩ : : = |
⟨simplex la stînga⟩ : : = ⟨⟨conexiune⟩
⟨simplex la dreapta⟩ : : = ⟨conexiune⟩⟩
⟨duplex⟩ : : = ⟨simplex la stînga⟩ ⟩ | ⟨simplex la stînga⟩, ⟨simplex la dreapta⟩
⟨half duplex⟩ : : = ⟨simplex la stînga⟩ / ⟨simplex la dreapta⟩
⟨cale de date⟩ : : = ⟨conexiune⟩ | ⟨simplex la stînga⟩ | ⟨simplex la dreapta⟩ | ⟨duplex⟩ | half duplex⟩ | ⟨nici o conexiune⟩
⟨definiţia căii⟩ : : = ⟨identificator de cale⟩ = ⟨structură⟩

Exemple

⟨identificator de cale⟩ : : = H ; H3 (pereche răsucită)
⟨timp în ns pentru un cuvînt⟩ : : = t ; 12 5 ; (1ms)
⟨biţi de date⟩ : : = 64 ; n
⟨biţi de adresă⟩ : : = 16 ; a
⟨număr de căi⟩ : : = 4 ; n
⟨mărimea unei cai⟩ : : = 4* ({64+16 ; 4*64 ; 64 ; 64+16
⟨magistrală de date⟩ : : = — ; $\frac{100}{4*\langle 64+16\rangle}$
⟨conexiune în cruce⟩ : : = x ; xxx
⟨conexiune⟩ : : = —H3 (pereche răsucită) — — — ; x x (reţea Banyan) x x ; —H3—
⟨nici o conexiune⟩ : : = |
⟨simplex la stînga⟩ : : = ←——— ; (x x ; ⟨— ; ⟨x
⟨simplex la dreapta⟩ : : = ————→ ; x x⟩ ; $\underset{150}{—}$⟩ ; x⟩
⟨duplex⟩ : : = ⟨—⟩ : $\underset{16}{←———}$, $\underset{32}{———→}$; ⟨$\underset{64}{————}$⟩
⟨half duplex⟩ : : = ⟨—/—⟩ ; ←——— H2— —/— —H3 ———→
⟨cale de date⟩ : : = : ; —⟩ ; — — ; x
⟨definire căi⟩ : : = H3 = {{$\underset{64}{———→}$, $\underset{8}{←}$ }/ $\underset{24}{←}$ }

A.6 Structuri

⟨unitate⟩ : : = ⟨E unitate⟩ | ⟨M unitate⟩ | ⟨calculator⟩ | ⟨literă mică⟩
⟨primar⟩ : : = ⟨unitate⟩ | {structură} ⟨conectivitate⟩ ⟨pipeline⟩
| ⟨structură paralelă⟩ ⟨pipeline⟩

⟨secundar⟩ : : = ⟨primar⟩ | ⟨factor⟩ ⟨primar⟩ | ⟨factor⟩ $\overline{\langle\text{primar}\rangle}$ | ⟨gol⟩
⟨structură⟩ : : = ⟨secundar⟩ | ⟨structură⟩ ⟨cale de date⟩ ⟨secundar⟩
⟨listă concurentă⟩ ; : = ⟨structură⟩ | ⟨listă concurentă⟩ | ⟨structură⟩
⟨listă secvenţială⟩ : = ⟨structură⟩ | ⟨listă secvenţială⟩ | ⟨structură⟩
⟨listă⟩ : : = ⟨listă concurentă⟩ | ⟨listă secvenţială⟩
⟨structură paralelă⟩ : : = $\{\langle\text{listă}\rangle\}^{\langle\text{conectivitate}\rangle}$
⟨definiţie⟩ : : = ⟨secundar⟩ = ⟨structură⟩

Exemple

⟨unitate⟩ : : = E ; M_{8*64} ; C[64$\overline{P}$] ; a
⟨primar⟩ : : = {−E−M−a) p ; {−E1−, E2} ; {E1/E2/E3} ; IO
⟨secundar⟩ : : = 16*32^2 $\overline{P}$; {3F, 2P}
⟨structura⟩ : : = −E−M−a ; {E−{−M1−M2−,−⟩}−−−M3}−a;
$\{32^2\overline{P}\}^{1-nn}$
⟨listă concurenţă⟩ : : = I, E, M
⟨listă secvenţială⟩ : : = 3E1/E2/E3
⟨structură paralelă⟩ : : = {I, E, M} ; {3E1/E2/E3} ; {−E−, | M−, −}
⟨definiţie⟩ : : = 3E1 = {E(+), E(*), E(÷)} ; E2={F(*)/F (+)/B} ;
C = I[64$\overline{P}$] $^{2-nn}_{1}$; M1= { − M $^{100}_{1K*}$ −,−→ }−M $^{(1ms)}_{1M*64}$

Index

OBJECTIVE C		
OCCAM		
OCCAM	alegere	
OCCAM	canale	
OCCAM	configurație	
OCCAM	constructori	
OCCAM	ieșire	
OCCAM	intrare	
OCCAM	procese	
OCCAM	sincronizarea proceselor	
OKI Ltd (Japonia)		
OMEN		
Oficiul meteorologic, Marea Britanie		
Operație de extragere-și-adunare		fetch-and-add (FA) operation
Operații elementare		elemental operations
Operator alfa		Alpha operator
Operator beta		Beta operator
Ortogonal	calculator	
Ortogonal	memorie	
Overhead	controlul buclei	
Overhead	de acces la memorie	
Overhead	de comunicație	
Overhead	programare	scheduling
Overhead	sincronizare	
PACK		
PAR		
PDPii		
PDP8		
PEPE (Burroughs)		
PHOENIX		
π, vezi performanța specifică		
PMOS		
PROLOG		
Pachet comutat		
Paralelism	al structurii	
Paralelism	algoritmic	
Paralelism	aparent	
Paralelism	conservare	